SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Dražen Mrvoš

Zagreb, 2012.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor:

Dr. sc. Mario Essert, dipl. ing.

Student:

Dražen Mrvoš

Zagreb, 2012.

Izjavljujem da sam ovaj rad izradio samostalno koristeći stečena znanja tijekom studija i navedenu literaturu.

Zahvaljujem se svojoj supruzi Marini koja mi je pomagala u najtežim trenucima tijekom studija. Također, zahvaljujem se svome mentoru, profesoru Mariu Essertu, na strpljenju, poticanju i pružanju znanja tijekom cijelog mog studija.

Dražen Mrvoš

SVEUČILIŠTE U ZAGREBU

FAKULTET STROJARSTVA I BRODOGRADNJE

Središnje povjerenstvo za završne i diplomske ispite

Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

DIPLOMSKI ZADATAK

Student: **Dražen MRVOŠ** Mat. br.: 0035161642

Naslov rada na hrvatskom jeziku: **Računalni program za označivanje i dokumentaciju**

Naslov rada na engleskom jeziku: **Computer Program for Marking Up and Documentation**

Opis zadatka:

Izrada tehničke dokumentacije nužna je u svim inženjerskim zanimanjima. Primjenom računala u tom dijelu struke bitno se olakšava izrada (i dopunjavanje) dokumentacije koja može biti u klasičnom obliku (*MS Word, LibreOffice Writer*, ...), *TeX* obliku ili nekom drugom. Danas postoje alati koji bitno olakšavaju izradu (pisane) dokumentacije, a također postoje i alati koji tu istu dokumentaciju prevode iz jednog oblika (formata) u drugi.

S druge strane, skeniranjem velikog broja dokumenata ili knjiga dolazi se do problema dokumentacije slika dokumenata (koje se, da bi bile dohvatljive, moraju mukotrpno opisivati). Problem se rješava na bolji način stavljanjem i spremanjem vizualnih oznaka na slike dokumenta, a da se pritom sam dokument ne mijenja.

Za realizaciju diplomskog zadatka potrebno je:

1. Osmisliti aplikaciju koja će za unesenu sliku (ili niz slika skeniranog materijala) omogućiti njen pregled, manipulaciju (rezanje), crtanje osnovnih elemenata na njoj (oznake, kružnice, strelice, ...) te dodavanje teksta (sa ili bez indeksa i potencija). Promjene se ne smiju spremati na sliku nego u bazu (primjerice SQLite) i to po slojevima (ako postoje različite kategorije označivanja);
2. Omogućiti postavljanje željenih oznaka (markera) na sliku;
3. Razlikovati nekoliko razina korisnika (administrator, pisanje i/ili čitanje, samo čitanje);
4. Proučiti i dokumentirati *ReST* markup jezik i *Python* biblioteku *Sphinx* za izradu dokumentacije;
5. Proučiti i demonstrirati *Pandoc* program koji služi za prevođenje iz jednog označiteljskog (markup) jezika u drugi.

Zadatak zadan:	Rok predaje rada:	Predviđeni datum obrane:
4. listopada 2012.	6. prosinca 2012.	12. – 14. prosinca 2012.

Zadatak zadao: Predsjednik Povjerenstva:

Prof.dr.sc. Mario Essert Prof. dr. sc. Franjo Cajner

Sadržaj

Popis slika

Popis tablica

Sažetak

Ovaj rad se bavi problematikom označavanja veće količine digitalnih slika, bez da se mijenja izvorna slika. U tu svrhu je osmišljena aplikacija koja korisniku omogućava jednostavno dodavanje oznaka. Oznake se dodaju u jedan sloj ili više njih, a korisnik odlučuje koji će od tih slojeva biti prikazan.

Tijekom izrade aplikacije, isproban je novi način izrade diplomskog rada korištenjem *Sphinx* biblioteke za *Python* koja omogućuje izradu funkcionalne i oku ugodne dokumentacije. Dokumentaciju se zatim može dobiti u *HTML*, *LaTeX* ili *epub* formatu.

Abstract

This paper deals with problems concerning marking large sets of computer images, without making any changes to original image. For that purpose, computer application that makes marking simple to use was developed. Markers can be added to one or more layers, and it's up to user to decide which layer (or layers) will be displayed.

During application development, a new way of making this kind of paper was tried by using *Sphinx Python* library that creates functional and eye-appealing documentation. Documentation can then be exported to *HTML*, *LaTeX* or *epub* format.

POGLAVLJE 1

Uvod

Prilikom skeniranja većeg broja stranica dolazi do problema pregledavanja skeniranog sadržaja. Obzirom da skeniranje ne vrši nužno osoba za koju je taj materijal namijenjen, obično krajnji korisnik mora provjeriti da li je skenirani materijal dobar ili ne. Primjerice, ukoliko se skenira cijela knjiga, neki dijelovi knjige mogu biti oštećeni, ili sami skenirani dokument može biti loš (primjerice mutan). Ukoliko se skenira stranica po stranica, većina problema se može zaobići, ali ako se automatizirano skenira - nužno je pregledati cijeli skenirani materijal. Iz tog razloga je kroz diplomski rad razvijena aplikacija koja bi taj proces olakšala.

Ono što program svakako mora sadržavati jest:

- Otvaranje slike i/ili čitavog niza slika
- Uvećavanje i pomicanje po prikazu slike (eng. *Zoom and Pan*)
- Dodavanje vizualnih elemenata po postojećoj slici (markeri)
- Crtanje različitih oznaka po slici (primjerice tekst)

Cijela aplikacija je razvijena u Python programskom jeziku, uz pomoć nekoliko biblioteka koje su opisane u kasnijim dijelovima rada.

Uz razvoj aplikacije se radilo i na novom načinu izrade tehničke dokumentacije. Naime, korištenjem *Sphinx* biblioteke za *Python*, moguće je na jednostavan način izraditi složenu, vizualno atraktivnu dokumentaciju. Dokumentacija se piše u reST markup jeziku, a pomoću *Sphinx* biblioteke se prevodi ili u *HTML* ili *LaTeX* jezik. Korištenjem *Pandoc* alata moguće je gotovo bilo koji markup jezik prevesti u gotovo bilo koji drugi, tako da se dokumentacija dobivena *Sphinx*-om može prevoditi u primjerice *epub* ili *mobi* format čime je prilagođena mobilnim uređajima ili e-čitačima knjiga (npr. *Kindle*).

Kroz diplomski rad će se opisati princip rada aplikacije, kao i njene mogućnosti. Uz to, što

jednostavnije će se pojasniti *reST* jezik i mogućnosti *Sphinx* biblioteke. Obzirom da je želja autora da i drugi (koji žele) mogu koristiti *Sphinx* bez bolnog podešavanja, opisati će se i način instalacije u *Windows* okruženju.

POGLAVLJE 2

Python i korištene biblioteke

Sadržaj poglavlja

Python je općenamjenski viši programski jezik sa vrlo opsežnom osnovnom bibliotekom. Glavne prednosti *Python*-a u usporedbi sa drugim programskim jezicima su:

- **Kraće vrijeme kodiranja** - u usporedbi sa jezicima *C*, *C++* i *Java* - *Python* za većinu algoritama ima 2-10 puta kraći kôd
- **Preglednost** - novi programski blokovi se označavaju dodatnim uvlakama, čime se programera prisiljava na strukturiranje programa
- **Lakoća korištenja** - *Python* se bezbolno može savladati u vrlo kratkom vremenskom periodu.

Iako je krajem devedestih godina prošlog stoljeća *Python* bio manje-više nepoznati jezik, danas je situacija mnogo drugačija. Obzirom da je kao skriptni jezik prilikom izvođenja sporiji od jezika kao što su *C* ili *Java*, sa današnjim je računalima to postalo gotovo nebitno pa je *Python* našao primjenu u mnogim poznatim tvrtkama i aplikacijama kao što su:

- **Google** - mnogi dijelovi tražilice (primarno dio za indeksiranje stranica - *spider-web*) su programirani upravo u *Python*-u

- **Yahoo!** - Yahoo! koristi *Python* u nekoliko svojih usluga - *Maps* i *Groups*
- **Civilization 4** - kompletni AI sustav je implementiran pomoću *Python*-a

2.1 Osnovne naredbe

Zbog lakše orijentacije u primjerima kôda, u narednih par redova su navedene najčešće korištene naredbe:

- **‘‘if‘‘** - uvjetno izvršava blok naredbi u kombinaciji sa naredbama **else** i **‘‘elif‘‘** koje se ne moraju nužno koristiti:

```
if uvjet:
        naredbe
elif uvjet:
        naredbe
else:
        naredbe
```

- **for** - za razliku od nekih programskih jezika, u *Python*-u `for` služi za iteriranje po bilo kojem nizu (primjerice listi ili stringu) i to u onom poretku po kojem se pojavljuju u nizu. Sve petlje u *Python*-u se mogu prekinuti naredbom `break`. Primjerice, kôd:

```
python = ['The','Flying', 'Circus']
for i in python:
        print python[i]
        break
```

će kao rezultat ispisati *‘The’*. Da se isti kôd pokrenuo bez naredbe `break`, ispisao bi se svaki element liste `python`. Primjerice, kôd napisan ispod će ispisivati svaki znak (*character*) stringa a:

```
a = 'FSB'
i = 0
while i<len(a):
        print a[i]
        i += 1
```

ili jednostavnije (odrađeno na *pythonovski* način):

```
a = 'FSB'
for each in a:
        print each
```

2.2 Rad sa iznimkama i greškama

Prilikom rada sa *Python* programskim jezikom, mogu se javiti 2 tipa grešaka: greške u sintaksi (eng. *syntax errors*) i iznimke (eng. *exceptions*).

2.2.1 Greške u sintaksi

Greške u sintaksi (ili greške prilikom parsiranja) su najčešći tipovi greške koji se dobivaju u procesu učenja *Python*-a:

```
>>> while True print 'Hello world'

File "<stdin>", line 1, in ?
        while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Prilikom javljanja ove greške, parser ponavlja liniju kôda koja nije ispravna i prikazuje *strelicu* na mjestu na kojem se najranije pronašla greška. Ova greška je uzrokovana (ili je barem na tom mjestu pronađena) izrazom prije same strelice: u navedenom primjeru, greška je pronađena pri ključnoj riječi `print`, budući da prije nje nedostaje dvotočka. Ime datoteke i broj linije u kojoj je greška pronađena se također ispisuju kako bi znali gdje pronaći sporno mjesto.

2.2.2 Iznimke (eng. *exceptions*)

Čak ako je naredba ili izraz sintaksno točna, ona može izazvati grešku prilikom izvršavanja. Greške koje se pronađu tijekom izvođenja, nazivaju se *iznimke* i nisu nužno kritične. U *Python*-u se rukovanje greškama (eng. *exception handling*) može vrlo brzo savladati.:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero

>>> 4 + fsb*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'fsb' is not defined

>>> '2' + 2
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Zadnja linija poruke o grešci govori o tome što se točno dogodilo. Iznimke su različitih tipova koji se ispisuju kao dio poruke, primjerice: `ZeroDivisionError`, `NameError` i `TypeError`. String koji se ispisuje kao tip iznimke je naziv ugrađene iznimke koja se dogodila. Ovo je točno za sve ugrađene iznimke, ali ne mora nužno biti točno za iznimke definirane od strane korisnika (iako ih je korisno definirati). Standardni nazivi iznimki su ugrađeni identifikatori (ključne riječi koje nisu rezervirane):

```
>>> TypeError = 2
>>> print TypeError
2
```

2.2.3 Rukovanje iznimkama (eng. *exception handling*)

Moguće je (i poželjno) napisati program koji će rukovati sa odabranim iznimkama. Sljedeći primjer traži korisnika da nešto unosi sve dok taj unos nije cjelobrojna vrijednost, ali također daje korisniku mogućnost da program prekine u bilo kojem trenutku (koristeći kombinaciju tipki `Control-C` ili koju god već kombinaciju tipki operativni sustav podržava). Ukoliko korisnik prekine program, program podiže iznimku (eng. *to raise an exception*) `KeyboardInterrupt`. Jednom kada se iznimka podigne, ona se prosljeđuje rutini za oporavak od iznimke (eng. *exception recovery routine*).

```
while True:
        try:
                x = int(raw_input("Unesite broj: "))
                break
        except ValueError:
                print "To nije valjani broj,  pokušajte ponovno..."
```

Naredba `try` (eng. `try` statement) funkcionira na sljedeći način:

- prvo se izvršava `try` blok (naredbe između `try` i `except`)
- ukoliko se ne podigne nikakva iznimka, `except` blok uvjet se preskače i izvršavanje `try` naredbe je gotovo
- ukoliko se prilikom izvršavanja podigne iznimka, ostatak tog bloka se preskače. Nakon toga, ukoliko se tip iznimke podudara sa iznimkom navedenom iza ključne riječi `except`, izvršava se blok `except` i nakon toga se kôd iza `try` naredbe dalje izvršava
- ukoliko se pojavi iznimka koja se ne slaže sa iznimkom navedenom iza `except` naredbe,

prenosi se sljedećim dijelovima `try` naredbe (primjerice ako ima više `except` blokova) i ako niti jedan ne odgovara tom tipu iznimke, ona se predstavlja kao `unhandled exception`, odnosno iznimka koja nije predviđena da se pojavi i izvršavanje kôda se u tom trenutku prekida

`Try` naredba dakle može imati više od jednog `except` uvjeta kako bi se mogle definirati radnje za različite iznimke. Prilikom izvršavanja, najviše će jedan *except* blok biti izvršen. Bitno je za naglasiti da se `except` dio izvršava samo unutar definirane `try` naredbe, što znači da se tako definirano rukovanje iznimkom izvršava samo ukoliko se iznimka pojavi prilikom izvršavanja `try` bloka. `Except` uvjet može sadržavati više iznimki na način da ih se sljedno navede unutar jednog *tuple*-a:

```
except (RuntimeError, TypeError, NameError):
        pass
```

Tuple je u ovom slučaju nužno koristiti jer se sintaksa `except ValueError, e:` koristila umjesto onoga što se danas zapisuje kao `except ValueError as e:` (opisano dalje u tekstu). Stara sintaksa je i dalje podržana zbog podrške skripti pisanih u starijim verzijama Python-a. To znači da izraz `except RuntimeError, TypeError` nije ekvivalentan izrazu `except (RuntimeError, TypeError):``nego izrazu ``except RuntimError as TypeError:`, a to vjerojatno nije ono što se želi.

U `try` bloku je moguće u zadnjem `except` izrazu ne navesti tip iznimke, što znači da će se blok izvršiti za svaki tip iznimke koji nije naveden u ranijim `except` blokovima. Ovo se treba koristiti sa velikim oprezom jer je na taj način moguće kamuflirati grešku u programu. No, može se i vrlo dobro iskoristiti kako bi se primjerice pokazala poruka o grešci i po potrebi podigla nova iznimka.

```
import sys

try:
        f = open('datoteka.txt')
        s = f.readline()
        i = int(s.strip())
except IOError as e:
        print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
        print "Podatke nije moguće konvertirati u integer."
except:
        print "Greška:", sys.exc_info()[0]
        raise
```

Uz `try ... except` naredbu se može po potrebi koristiti i `else` uvjet, koji kada je prisutan mora pratiti sve `except` naredbe. Vrlo je koristan za kôd koji se mora izvršiti ukoliko

`try` uvjet ne podigne iznimku:

```
for arg in sys.argv[1:]:
        try:
                f = open(arg, 'r')
        except IOError:
                print 'ne mogu otvoriti', arg
        else:
                print arg, 'ima', len(f.readlines()), 'linija'
                f.close()
```

Korištenje `else` uvjeta je mnogo bolje nego dodavanje kôda u `try` uvjet zbog toga toga što onemogućuje slučajno podizanje iznimke kôdom koji ne bi trebao biti zaštićen `try ... except` naredbom.

Kada dođe do iznimke, ona može imati i neku vrijednost što se naziva *argument* iznimke. Sama prisutnost i tip argumenta ovisi o tipu iznimke.

U `except` naredbi, iza naziva iznimke se može navesti varijabla. Varijabla je vezana za instancu iznimke sa argumentima spremljenim u `instance.args`. Zbog jednostavnijeg korištenja, instanca iznimke definira `__str__()` tako da se argumenti mogu ispisati direktno bez da se referencira `.args`.

Moguće je i instancirati iznimku prije nego je se podigne i dodati joj željene atribute:

```
>>> try:
...     raise Exception('Miki', 'Pajo')
... except Exception as inst:
...     print type(inst)     # instanca iznimke
...     print inst.args      # arguments pohranjeni u .args
...     print inst
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y

<type 'exceptions.Exception'>
('Miki', 'Pajo')
('Miki', 'Pajo')
x = Miki
y = Pajo
```

Ukoliko iznimka ima argument, ispisuje se kao posljednji dio ('detalj') poruke o neočekivanoj iznimci.

Rukovanjem iznimki se ne postiže samo to da se njima rukuje ukoliko se pojave u samome `try` bloku, nego ako se pojave i unutar funkcije koja se poziva (čak i indirektno) unutar `try` naredbe. Primjerice:

```
>>> def ovo_nece_uspjeti():
...     x = 1/0
...
>>> try:
...     ovo_nece_uspjeti()
... except ZeroDivisionError as detalj:
...     print 'Handling run-time error:', detalj
...
Handling run-time error: integer division or modulo by zero
```

2.2.4 Podizanje iznimki

`Raise` naredba omogućuje programeru da se određena iznimka obavezno pojavi. Primjerice:

```
>>> raise NameError('Bok')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Bok
```

Sami argument `raise` inducira podizanje iznimke. To mora biti ili instanca iznimke ili klasa iznimke (klasa koja proizlazi iz `Exception` klase).

Ukoliko je potrebno odrediti da li je iznimka podignuta ali bez namjere da se njome rukuje, pojednostavljeni oblik `raise` naredbe omogućuje da se iznimka ponovno podigne:

```
>>> try:
...     raise NameError('Bok')
... except NameError:
...     print 'Evo iznimke!'
...     raise
...
Evo iznimke!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: Bok
```

2.2.5 Iznimke definirane od strane korisnika

Programi mogu imenovati vlastite iznimke definiranjem nove klase iznimke. Iznimke bi obično trebale biti izvedene iz klase `Exception`, direktno ili indirektno. Primjer:

```
>>> class MyError(Exception):
...     def __init__(self, value):
```

```
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'Doslo je do moje pogreske, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('ups!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'ups!'
```

U gornjem primjeru, zadani `__init__` je promijenjen na vlastiti. Nova radnja te funkcije jednostavno stvara atribut `value`. Ovo zamjenjuje zadano ponašanje kreiranja `args` atributa.

Mogu se definirati klase iznimki koje rade sve što i druge klase, ali se obično pojednostavljuju, a često sadrže samo dovoljan broj atributa koji omogućuju davanje informacija o pogrešci kako bi se moglo rukovati sa njima. Ako se kreira modul koji može podići nekoliko specifičnih grešaka, praksa je da se napravi osnovna klasa iz koje će se dalje kreirati specifične potklase za različite greške.

Većina iznimki se definira tako da im naziv završava sa `Error`, slično nazivima standardnih (ugrađenih) iznimki. Većina standardnih modula u *Python*-u imaju definirane vlastite iznimke u svrhu prijave greške koja se pojavila tijekom izvođenja određene funkcije.

2.3 Kontrola toka programa

2.3.1 if naredba

Ova naredba se uči među prvima u svim programskim jezicima. Primjer:

```
>>> x = int(raw_input("Unesi neki broj: "))
Unesi neki broj: 42
>>> if x < 0:
...      x = 0
...      print 'Negativnom broju je nova vrijednost - 0'
... elif x == 0:
...      print 'Nula'
... elif x == 1:
...      print 'Jedan'
... else:
```

```
...       print 'Veći od jedan'
...
More
```

Dijelova `elif` može biti od nijednog do neodređeno mnogo, `else` dio nije nužno koristiti. Ključna riječ *elif* je skraćenica za 'else if' i korisna je kako bi se izbjeglo pojavljivanje dugačkih uvlaka. Slijed naredbi *if ... elif ... elif ...* je zamjena za naredbe `switch` ili `case` koje se obično pojavljuju u drugim programskim jezicima.

2.3.2 for naredba

`for` naredba je u *Pythonu* malo drugačija od klasičnih `for` petlji u *C*-u ili *Pascal*-u. Umjesto da se uvijek iterira aritmetičkim slijedom brojeva (npr. kao u *Pascal*-u), ili da se korisniku dopusti da upravlja sa korakom iteracije i uvjetom zaustavljanja petlje (kao u *C*-u), *Pythonova* `for` naredba iterira elemente bilo kojeg niza (npr. liste ili stringa), u onom poretku u kojem se pojavljuju u nizu. Primjer:

```
>>> # Duljina niza stringova:
... a = ['mačka', 'zastori', 'kaos']
>>> for x in a:
...     print x, len(x)
...
mačka 5
zastori 7
kaos 4
```

Nije sigurno mijenjati niz po kojem se iterira u petlji (ovo se može dogoditi isključivo sa tipovima koji su promjenjivi, poput listi). Ukoliko je potrebno mijenjati listu po kojoj se iterira, mora se iterirati po kopiji:

```
>>> for x in a[:]: # stvara se kopija cijele liste
...    if len(x) > 5: a.insert(0, x)
...
>>> a
['zastori', 'mačka', 'zastori', 'kaos']
```

2.3.3 Naredba range

Ukoliko je potrebno iteraciju provesti po slijedu brojeva, vrlo korisna naredba je `range` koja generira aritmetički niz brojeva kao listu:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Broj naveden kao argument nikada nije dio generirane liste; `range(10)` generira listu sa 10 vrijednosti. Kako bi naredba bila fleksibilnija, moguće je zadati granice najvećeg i najmanjeg generiranog broja, kao i korak iteracije:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Kako bi se iteracija provela po indeksima liste, moguće je kombinirati naredbe `range` i `len`:

```
>>> a = ['Per', 'aspera', 'ad', 'astra']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Per
1 aspera
2 ad
3 astra
```

2.3.4 `break` i `continue` naredbe i `else` uvjeti u petljama

Naredba `break` prekida najbližu `for` ili `while` petlju koja ju obuhvaća.

S druge strane, naredba `continue` nastavlja sa iteracijom u petlji.

Petlje mogu imati `else` uvjet koji se izvršava kada više nema elemenata po kojima se iterira (u slučaju `for` petlje) ili kada zadani uvjet postane `False` (u slučaju `while` petlje):

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, '=', x, '*', n/x
...             break
...     else:
...         # petlja je završena bez da je nađen djelitelj
...         print n, 'je prosti broj'
...
2 je prosti broj
3 je prosti broj
```

```
4 = 2 * 2
5 je prosti broj
6 = 2 * 3
7 je prosti broj
8 = 2 * 4
9 = 3 * 3
```

U ovoj primjeni je `else` uvjet sličniji `else` uvjetu kod `try` naredbe nego kod `if` naredbi: kod `try` naredbe se `else` uvjet izvrši kada ne dođe do iznimke. Slično tome, `else` uvjet kod petlji se izvrši kada se cijeli uvjet izvrši bez ijedne `break` naredbe.

2.3.5 *pass* naredba

Naredba `pass` je vrlo korisna a ništa ne radi. Može se koristiti kada je nužno izvršiti barem jednu naredbu iako to nije potrebno u samome programu:

```
>>> while True:
...     pass  # Čekaj prekid tipkovnicom (Ctrl+C)
...
```

Ova se naredba može koristiti i kod definiranja minimalne klase:

```
>>> class PraznaKlasa:
...     pass
...
```

Vrlo bitna primjena ove naredbe je prilikom početnog definiranja nove funkcije ili klase kada nismo sigurni što će funkcija sve sadržavati ali nam je trenutno zbog drugih dijelova kôda bitno da barem postoji. Primjer, gore navedenoj klasi `PraznaKlasa`, dodaje se sljedeća metoda:

```
>>> def initlog(*args):
...     pass   # Dodati kasnije!
...
```

Ukoliko se sada pozove metoda `PraznaKlasa.initlog()`, ništa se neće dogoditi. Kasnije se, kada to programeru vrijeme dopusti, metoda `initlog` definira po potrebi.

2.3.6 Definiranje funkcija

```
>>> def fib(n):    # Fibonaccievi brojevi do n
...     """Ispisuju se Fibonaccievi brojevi do n."""
...     a, b = 0, 1
```

```
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Pozivanje definirane funkcije:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ključna riječ `def` definira funkciju `definiranje`. Nakon nje mora slijediti ime funkcije (`fib`) i formalni parametri u zagradi (`(n)`). Od sljedeće linije kôda kreću sve naredbe koje spadaju pod definiranu funkciju i one moraju biti uvučene (kao i u gornjem primjeru).

Prva naredba u tijelu funkcije može biti string koji opisuje funkciju (*documentation string* - *docstring*). Danas postoje alati koji koriste *docstring* kako bi automatizirano napravili online ili tiskanu dokumentaciju, ili u najmanju ruku omogućiti korisniku da interaktivno prolazi kroz kôd. Iz tog razloga je dobra navika uključivati *docstring* u kôd.

Izvršavanjem funkcije se uvodi nova tablica simbola (eng. *symbol table*) koja se koristi za pohranu lokalnih varijabli funckije. Preciznije, sva pridruživanja varijabli u funkciji se spremaju u lokalnu tablicu simbola tako da se reference na varijable prvo traže u lokalnim tablicama, zatim u lokalnim tablicama funkcija koje su pozvale tu funkciju, globalnim tablicama simbola i na kraju u tablici ugrađenih naziva. Iz tog razloga nije moguće iz funkcija direktno pristupiti globalnim varijablama (moguće je uz pomoć funkcije `global`).

Parametri (argumenti) funkcije se uvode u lokalnu tablicu simbole pozvane funkcije onda kada ju se pozove te se tako argumenti prosljeđuju koristeći pozivanje po vrijednosti (eng. *call by value*). Kada jedna funkcija pozove drugu, za taj poziv se stvara nova tablica simbola.

Definicija funkcije uvodi naziv funkcije u trenutnu tablicu simbola. Vrijednost naziva funkcije se sprema u obliku koji interpreter prepoznaje kao funkcija definirana od strane korisnika. Ta se vrijednost može pridruživati drugim nazivima koji se tada također mogu koristiti kao funkcija:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Ovdje treba naglasiti da bi po definicijama iz ostalih jezika `fib` bila procedura a ne funkcija jer ne vraća nikakvu vrijednost. U stvarnosti je kod *Python*-a ovo malo drugačije jer svaka funkcija vraća neku vrijednost. U ovom slučaju funkcija vraća vrijednost `None` ali ju interpreter ne prikazuje. To se može vidjeti koristeći naredbu `print`:

```
>>> fib(0)
>>> print fib(0)
None
```

Jednostavno je napisati funkciju koja će vraćati listu Fibonaccievih brojeva umjesto da ispisuje svaki broj:

```
>>> def fib2(n): # vraća Fibonacciev niz do n
...     """Vraća listu koja sadrži Fibonacciev niz do n."""
...     rezultanta = []
...     a, b = 0, 1
...     while a < n:
...         rezultanta.append(a)    # pogledaj ispod
...         a, b = b, a+b
...     return rezultanta
...
>>> f100 = fib2(100)     # pozovi funckiju
>>> f100                 # ispiši rezultat
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Gornji primjer demonstrira još neke nove značajke *Python*-a:

- naredba `return` vraća vrijednost iz funkcije. Bez argumenta `return` vraća vrijednost `None` kao i izvođenje procedure (funkcija `fib`).
- naredba `result.append(a)` poziva *metodu* objekta liste `result`. Metoda je funkcija koja propada objektu i naziva se `obj.imemetode` gdje je `obj` neki objekt, a `imemetode` je naziv metode koja je definirana za taj tip objekta. Različiti tipovi definiraju različite metode. Metode različitih tipova mogu imati isti naziv bez da zadaju glavobolju oko toga koja što radi. Metoda `append` je definirana za sve objekte listi; ona dodaje novi element na kraj liste. U ovome primjeru je taj izraz identičan izrazu `result = result + [a]`, ali je puno praktičniji i jednostavniji.

2.3.7 Detaljnije o definiranju funkcija

Moguće je definirati funkcije sa varijabilnim brojem argumenata. Postoje tri oblika koji se mogu kombinirati.

Zadane vrijednosti argumenata

Ovo je najkorisniji oblik definiranja zadanih vrijednosti. Ovo omogućuje da se funkcija pozove sa manje argumenata nego je definirano. Primjerice:

```
def DaNe(unos, pokusaji=4, prigovor='Da ili ne, molim!'):
        while True:
                ok = raw_input(unos)
                if ok in ('d', 'da'):
                   return True
                if ok in ('n', 'ne'):
                   return False
                pokusaji = pokusaji - 1
                if pokusaji < 0:
                   raise IOError('korisnikError')
                print prigovor
```

Ova se funkcija može pozvati na nekoliko načina:

- zadajući samo obavezni argument: `DaNe('Jesi dobro?')`
- zadajući samo jedan od neobaveznih argumenata: `DaNe('Sigurno to želiš izbrisati?', 2)`
- zadajući sve argumente: `DaNe('Sigurno to želiš izbrisati?', 2, 'Da ili ne samo!')`

Ovaj primjer uvodi novu ključnu riječ `in`. Ona provjerava da li se u nizu pojavljuje određena vrijednost.

Zadane vrijednosti se provjeravaju samo prilikom definiranja funkcije:

```
>>> i = 5
...
>>> def f(arg=i):
...     print arg
>>>
>>> i = 6
>>> f()
>>> 5
```

Potrebno je obratiti pozornost na to da se zadana vrijednost evaluira samo jednom. Ovo čini veliku razliku između slučaja kada je zadana vrijednost promjenjivi (eng. *mutable*) tip poput liste, rječnika ili instanci klasa.

Primjer kada je zadana vrijednost promjenjivog tipa:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
```

```
print f(3)
```

Gore navedeni kôd će ispisati:

```
[1]
[1, 2]
[1, 2, 3]
```

Ukoliko je potrebno izbjeći da se vrijednost zapravo prosljeđuje iz poziva u poziv, spretnije je zapisati funkciju poput sljedeće:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Ključne riječi kao argumenti

U prethodnom dijelu je opisano definiranje funkcije sa zadanim argumentima, gdje su svi zadani argumenti različitih tipova i dovoljno je bilo pozvati funkciju sa vrijednosti argumenta/argumenata. No, funkcija se može pozvati i korištenjem ključnih riječi za argumente. Primjer:

```
def zadaca(broj_stranica, predmet="OP", autor="Pajo", status="Jao"):
        print "Evo pisem baš zadacu iz ", predmet
        print "Mislim da ti ", broj_stranica, " nije dovoljno ..."
        print "A tko je ", autor, "?"
        print "Trenutna faza je, ", status
```

Funkcija zadaca prima jedan obavezan argument (`broj_stranica`) i tri neobavezna argumenta (`predmet`, `autor` i `status`). Ova se funkcija može pozvati na bilo koji od sljedećih načina:

```
zadaca(10)                              # 1 pozicijski argument
zadaca(broj_stranica=10)                # 1 argument sa ključnom riječi
zadaca(broj_stranica=20, predmet="terma") # 2 argumenta sa k. riječi
zadaca(10, "terma", "dobar kolega")       # 3 poz. argumenta
zadaca(10, "fluidi")                    # 1 poz. arg., 1 sa k. riječi
```

Sljedeći pozivi bi bili neispravni (u komentarima su navedene dobivene greške):

```
zadaca()                        # required argument missing
zadaca(broj_stranica=5.0, "neki")   # non-keyword argument after
```

```
                                    # keyword argument
zadaca(5, broj_stranica=10)  # duplicate value for the same argument
zadaca(zabavno="ludilo")     # unknown keyword argument
```

Kao što se može primijetiti, u prvom primjeru nedostaje obavezan argument, u drugom se nakon argumenta sa ključnom riječi nalazi argument bez ključne riječi, u trećem su zadane dvije vrijednosti za isti argument i u zadnjem primjeru je zadan argument sa nedefiniranom ključnom riječi.

Prilikom pozivanja funkcije, argumenti sa ključnom riječi moraju pratiti pozicijske argumente. Svi argumenti sa ključnom riječi moraju odgovarati jednom od argumenata koje funkcija prihvaća (primjerice, argument `zabavno` nije odgovarajući argument za funkciju zadaca), i njihov poredak nije bitan. Ovo uključuje i obavezne argumente. Sljedeći primjer pokazuje što se dešava kada se ovo pravilo prekrši:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Kada zadnji parametar u definiranju funkcije sadrži oblik `**naziv`, poprima rječnik (*dictionary* tip) koji sadrži sve argumente sa ključnom riječi izuzev onih koji odgovaraju formalnim parametrima. Ovo se može kombinirati sa formalnim parametrima oblika `*naziv` koji poprima tuple koji sadržava pozicijske argumente iza niza formalnih parametara. `*naziv` se obavezno mora navesti prije `**naziv`.

Ako se definira funkcija na sljedeći način:

```
def cheeseshop (kind, *arguments, **keywords):
   print "Imate li ", kind, "?"
   print "Zao mi je, nemamo vise ", kind
   for arg in arguments:
           print arg
   print "-" * 40
   keys = sorted(keywords.keys())
   for kw in keys:
           print kw, ":", keywords[kw]
```

Ako se pozove na sljedeći način:

```
cheeseshop("Python", "neki argument",
                  "I jos neki argument",
                  mjesto='Zagreb',
                  kupac="Ivan Ivanic")
```

Rezultat će biti:

```
Imate li  Python ?
Zao mi je, nemamo vise  Python
neki argument
I jos neki argument
----------------------------------------
kupac : Ivan Ivanic
mjesto : Zagreb
```

Bitno je primijetiti da se lista argumenata sa ključnom riječi stvara sortiranjem rezultata metode `keys()` pozvanom nad rječnikom ključnih riječi. Ako se ovo ne bi napravilo, redoslijed ispisa argumenata bi bio nedefiniran.

Lambda izrazi

Ova mogućnost je bila vrlo popularna u jezicima usmjerenim na funkcijsko programiranje, poput *Lisp*-a. Iz tog razloga je dodana i u *Python* kako bi se mogle napraviti male, anonimne funkcije. Sljedeća takva funkcija vraća zbroj njena dva argumenta: `lambda a, b: a+b`. Lambda izrazi se koriste kad god su potrebni funkcijski objekti. Sintaktički su ograničeni na jedan izraz. Poput definiranih ugnježđenih funkcija, lamba izrazi mogu referencirati varijable sadržanog opsega:

```
>>> def uvecaj(n):
...     return lambda x: x + n
...
>>> f = uvecaj(42)
>>> f(0)
42
>>> f(1)
43
```

Stringovi za dokumentaciju

Zasad su konvencije oko sadržaja i formatiranja dokumentacijskih stringova neslužbene ali se svejedno mnogi pridržavaju određenih savjeta.

Prva linija bi uvijek trebala biti kratka i sadržavati sažeti opis funkcije objekta. Zbog sažetosti,

ne bi se trebalo doslovce navoditi ime ili tip objekta jer se do toga može doći na druge načine (osim ako je baš sami naziv ujedno i najbolji opis funkcije objekta). Ova linija bi uvijek trebala započinjati sa velikim slovom i završavati sa točkom.

Ukoliko dokumentacijski string sadrži više linija, druga bi trebala biti prazna tako da se i vizualno odvoji sažetak od ostatka opisa. Dio iza bi trebao biti jedan ili više odlomaka koji opisuju način pozivanja objekta, posljedice i sl.

Python-ov parser ne uklanja (eng. *strip*) uvlake iz višelinijskih stringova, tako da alati za izradu dokumentacije to moraju učiniti sami ukoliko se to želi. Za to također postoji dogovor. Prva linija koja nije prazna *nakon* prve linije stringa određuje razinu uvlaka za cijeli dokumentacijski string. Nakon toga se iz svih linija stringa uklanjaju prazna mjesta (eng. *whitespace*) koja su po dužini jednaki dužini uvlake. Linije koje imaju manje uvlake od referentne se ne bi smjele pojaviti, ali ako se i pojave - sva vodeća prazna mjesta se uklanjaju. Ekvivalentnost praznih mjesta bi se trebala provjeriti nakon što se uvlake transformiraju u oblik praznih mjesta (obično je uvlaka jednaka 8 praznih mjesta).

Primjer:

```
>>> def moja_funkcija():
...     """Ne radi ništa, ali barem je dokumentirana.
...
...     Ozbiljno, funkcija ništa ne radi.
...     """
...     pass
...
>>> print moja_funkcija.__doc__
Ne radi ništa, ali barem je dokumentirana.

    Ozbiljno, funkcija ništa ne radi.
```

2.4 Moduli

Ukoliko se *Python*-ov interpreter ugasi i zatim ponovno upali, sve definicije (definirane funkcije i varijable) - nestaju. Iz tog razloga, a pogotovo kada se radi složeniji i opsežniji program, bolje je u nekome programu za uređivanje teksta pripremiti datoteku za unos u interpreter i zatim pokretati tu datoteku kao unos. To se još naziva *izrada skripti*. Kako se program povećava, lakše je održavati kôd ukoliko se on razdvoji u nekoliko datoteki. Osim toga, moguće je da će zgodno doći mogućnost da se neka funkcija izvrši bez da se prije toga kopira u svaki program.

U *Python*-u postoji jednostavan način da se definicije spreme u zasebnu datoteku i zatim koristi kao skripta ili interaktivna instanca interpretera. Takva se datoteka naziva *modul*; definicije iz

modula se mogu uvesti (eng. *import*) u druge module ili u glavni (*main*) modul (skup varijabli kojima se može pristupiti iz skripte izvršene na najvišoj razini ili u kalkulatorskom načinu rada).

Modul je datoteka koja sadrži *Python* definicije i naredbe. Naziv datoteke je naziv modula sa dodanim sufiksom *.py*. Unutar modula, naziv modula (kao string) je dostupan kao vrijednost globalne varijable `__name__`. Primjerice, pomoću odabranog programa za uređivanje teksta se u trenutnom *Python* direktoriju napravi datoteka naziva *fibo.py* sa sljedećim sadržajem:

```
# Modul Fibonaccijevih brojeva

def fib(n):    # ispisuje Fibonaccijev niz do n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # vraca Fibonaccijev niz do n
    rezultat = []
    a, b = 0, 1
    while b < n:
        rezultat.append(b)
        a, b = b, a+b
    return rezultat
```

Nakon toga se iz *Python* interpretera uveze taj modul pomoću sljedeće naredbe:

```
>>> import fibo
```

Ovo ne uvodi nazive funkcija iz `fibo` direktno u trenutnu tablicu simbola; ono samo unosi naziv modula `fibo`. Koristeći naziv modula može se pristupiti funkcijama:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Ukoliko će se neka funkcija često koristiti, moguće ju je pridružiti lokalnom nazivu:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

2.4.1 Paketi

Jednostavno rečeno, paketi su metoda strukturiranja naziva *Python* modula koristeći nazive modula "sa točkom". Primjerice, naziv modula `A.B` označuje podmodul naziva `B` u paketu sa nazivom `A`. Jednako kao što moduli štede autore muke oko preklapanja naziva globalnih varijabli, tako ih korištenje naziva sa točkama štedi muka oko naziva paketa sa više modula ili kao *NumPy* ili *Python Imaging Library (PIL)* brige oko naziva modula jednog i drugog.

Ako se pretpostavi da se želi kreirati kolekcija modula (paket) za rukovanje različitim zvučnim datotekama i zapisima. Postoji mnogo različitih formata (obično se razlikuju po ekstenziji, primjerice `.wav`, `.mp3`, `.aiff`, ...) pa bi bilo potrebno kreirati i održavati rastuću kolekciju modula za konverziju između različitih formata. Postoji također i veliki broj različitih operacija koje bi se primjenile na zvučni zapis (miksanje, dodavanje jeke, ...), tako da bi se zapisivao i beskrajan niz modula koji bi definirali takve operacije. Primjer takve strukture bi bio:

```
sound/                          Glavni paket
      __init__.py               Inicijalizacija paketa za zvuk
      formats/                  Podpaket za pretvorbu formata
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Podpaket za zvučne efekte
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Podpaket filtera
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

`__init__.py` datoteka je nužna kako bi *Python* tretirao direktorije na način da sadrže pakete; ovo se radi kako bi se izbjeglo da direktoriji sa čestim imenima poput `string` ne bi slučajno sakrili valjane module koji se nalaze kasnije u putanji pretrage direktorija. U najjednostavnijem slučaju, `__init__.py` može biti prazna datoteka, ali može i izvršavati inicijalizacijski kôd za paket ili postaviti `__all__` varijable.

Korisnici paketa mogu iz njega uvesti pojedinačne module:

```
import sound.effects.echo
```

Navedeno učitava podmodul `sound.effects.echo`. Prilikom pozivanja njegovog sadržaja, cijeli se naziv modula mora navesti:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alternativni način uvoza podmodula bi bio:

```
from sound.effects import echo
```

Ovo učitava podmodul `echo` što omogućuje njegovo pozivanje bez korištenja cijeloga prefiksa paketa pa se može koristiti na sljedeći način:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Još jedna varijanta je direktan uvoz željene funkcije ili varijable:

```
from sound.effects.echo import echofilter
```

Na ovaj način je funkcija `echofilter` direktno dostupna za korištenje:

```
echofilter(input, output, delay=0.7, atten=4)
```

Valja napomenuti da prilikom korištenja `from paket import predmet`, *predmet* može biti podmodul (ili potpaket) paketa, ili neki drugi naziv definiran u paketu poput funkcije, klase ili varijable. `import` naredba prvo provjerava da li je predmet definiran u paketu; ako nije, pretpostavlja da je to modul i pokušava ga očitati. Ukoliko takav modul ne nađe, podiže se iznimka `ImportError`.

Suprotno ovome, kada se koristi sintaksa kao `import predmet.potpredmet.potpotpredmet`, svaki predmet izuzev zadnjeg mora biti paket; zadnji predmet može biti modul ili paket ali ne može biti klasa, funkcija ili varijabla definirana u predmetu prije.

2.4.2 Uvoz * iz paketa

Dobro pitanje jest što se dešava kada korisnik upiše `from sound.effects import *`? U nekom idealnom svijetu, čovjek bi se nadao da će na neki način pronaći sve podmodule koji su prisutni u paketu i sve ih uvesti. Ovo bi trajalo jako dugo i uvoz podmodula bi imao neželjenih posljedica koje bi se trebale dogoditi samo kada se eksplicitno uveze podmodul.

Jedino rješenje je da autor paketa napravi eksplicitni sadržaj paketa. `import` naredba koristi sljedeći princip rada: ukoliko `__init__.py` datoteka paketa ima definiranu listu naziva `__all__`, ona se uzima kao lista naziva modula koji će se uvesti kada se upotrijebi naredba `from paket import *`. Na autoru paketa je da ažurira tu listu kada dođe do promjena u paketu. Autori paketa modu odlučiti i da to neće podržavati, odnosno da ne vide smisao u korištenju `import \*` za njihov paket. Primjerice, datoteka *sounds/effects/__init__.py* može sadržavati sljedeći kod:

```
__all__ = ["echo", "surround", "reverse"]
```

Ovo bi značilo da će `from sound.effects import *` uvesti samo tri navedena podmodula iz `sound` paketa.

Ukoliko `__all__` nije definiran, naredba `from sound.effects import *` *ne* uvozi sve podmodule iz paketa `sound.effects` nego samo osigurava da je paket `sound.effects` uvezen (i da je eventualno izvršen inicijalizacijski kôd iz `__init__.py`) i zatim uvozi sve definirane nazive iz paketa. To uključuje bilo koji definirani naziv (i eksplicitno učitani podmodul) iz datoteke `__init__.py`. Također uključuje sve podmodule paketa koji su eksplicitno učitani prethodnim `import` naredbama:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

U gornjem primjeru, `echo` i `surround` moduli se uvoze u trenutni prostor imena (eng. *namespace*) jer su definirani u `sound.effects` paketu kada se izvršava `from...import` naredba. (Ovo funkcionira i kada je `__all__` definiran)

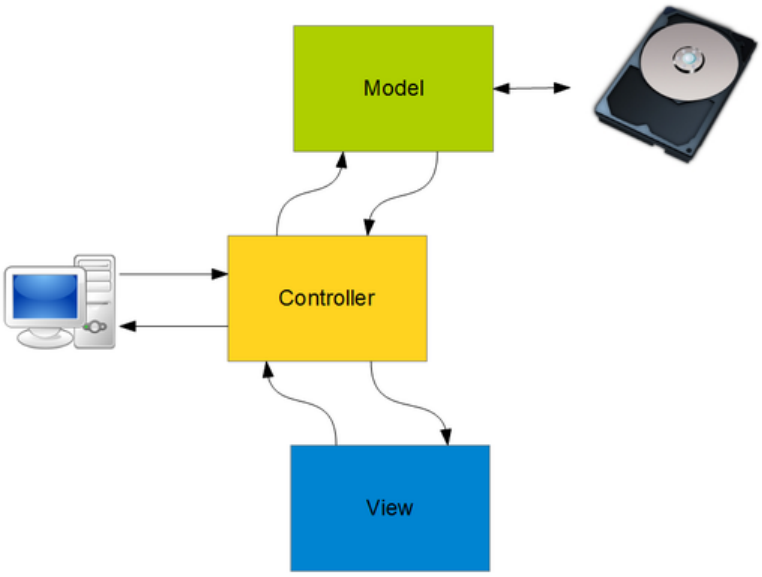
Iako neki moduli imaju definiranu `__all__` listu i time podržavaju `import *`, i dalje se smatra lošom praksom ga koristiti (lošija preglednost i koda).

Valja napomenuti da nema ničeg lošeg sa korištenjem `from paket import neki_podmodul`. U stvari, ovo je preporučljiva notacija osim ako uvezeni modul treba koristiti podmodula sa istim nazivom iz različitih paketa.

2.5 MVC arhitektura

MVC (*Model-View-Controller*) je arhitektura koja odvaja prikaz informacija i korisnikovu interakcija sa njima. *Model* u praksi sadrži pristup bazi podataka i definiciju svih funkcija potrebnih za rad sa njom. *Controller* upravlja korisničkim unosima, a *View* je prikaz podataka (npr. dijagram, polje teksta i sl.). Osnovna ideja MVC-a je lakša ponovna uporaba kôda i razdvajanje

razvoja.



Slika 2.1: Struktura MVC klase

2.5.1 Interakcija pojedinih dijelova

Osim što MVC dijeli aplikaciju na 3 osnovna dijela, on također definira i njihovo međudjelovanje.

- **Controller** - na temelju korisnikovih unosa mijenja prikaz modela ili modelu javlja da napravi *update*. Primjerice, korisnik koristi neki alat poput *Word* procesora. Ukoliko korisnik mišem *scroll*-a, *Controller* mora narediti promjenu prikaza otvorenog dokumenta. Ukoliko korisnik pobriše i spremi dio dokumenta, *Model* dio mora na sebi odraditi *update*
- **Model** obavještava *view* dijelove i *controller* da je došlo do neke promjene. Ta obavijest omogućuje *view* dijelovima da osvježe prikaz, a *controlleru* da promijeni dostupan skup naredbi. Postoji pasivna izvedba MVC-a koja ne sadrži ove obavijesti jer ih ili ne treba ili ih razvojna platforma ne podržava.
- **View** od *modela* zahtijeva informacije koje su mu potrebne da generira prikaz

Ova arhitektura je gotovo nezaobilazna kod WEB aplikacija, gdje se mora razdvojiti klijentski i poslužiteljski dio. U tom slučaju je *View* ono što se vidi na zaslonu računala klijenta, *Controller* je primjerice PHP ili Python skripta, a *Model* je neka vrsta baze podataka (npr. *MySQL*).

POGLAVLJE 3

Korištene biblioteke

Sadržaj poglavlja

U sljedećih nekoliko stranica su opisane korištene biblioteke koje ne dolaze uz instalaciju *Python*-a.

3.1 SetupTools - Easy Install

Easy Install je *python* modul koji dolazi u sklopu paketa *SetupTools*. Radi se o modulu koji pojednostavljuje instalaciju ostalih *Python* modula. Jednom kada se paket instalira sa službene stranice, moguće je početi koristiti `easy_install`. Ukoliko je putanja *Python*-a (i njegovog poddirektorija *Scripts*) unešena u sistemsku varijablu *PATH* (Windows, Linux distribucije bi to već trebale imati podešeno), dovoljno je otvoriti konzolu (Windows: Command prompt, Linux: Terminal ili koji već dolazi sa distribucijom) i upisati:

```
easy_install ime_paketa
```

Ukoliko putanja nije podešena, nakon što se pokrene konzola, prvo je potrebno promijeniti trenutni direktorij u instalacijski direktorij *Python*-a, a zatim u poddirektorij *Scripts* (npr. `cd`

`c:\Python27\Scripts`). Nakon toga je procedura ista kao i u gornjem primjeru. Ovakva instalacija je moguća isključivo ako ju modul podržava. Primjerice, *wxPython* modul trenutno nije moguće na ovaj način instalirati.

3.2 Python Imaging Library - PIL

PIL modul *Python*-u daje mogućnost obrade digitalnih slikovnih datoteki. Biblioteka podržave mnogo različitih formata. U kombinaciji sa *Python*-om je *PIL* izrazito moćan alat kad god se treba napraviti skripta kojom će se obrađivati veliki broj (primjerice) fotografija.

PIL sam po sebi podržava funkcionalnost kao što je filtriranje korištenjem konvolucijske matrice ili prebacivanje iz jednog prostora boja u drugi (primjerice RGB u BGR). Osim toga, slikovne datoteke je moguće rezati, mijenjati im veličinu, rotirati, ... Uz sve to, jednostavno je dobiti histogram iz kojeg se mogu izvući i neki statistički podaci o slici.

Glavna klasa *PIL*-a je `Image`. Ta se klasa može instancirati na nekoliko načina: čitanjem iz datoteke, procesiranjem drugih slikovnih datoteki ili kreiranjem slike iz nule. Primjerice, ukoliko se želi otvoriti slika `python.png`, ispisati njene dimenzije i spremiti je kao jpg datoteku - dovoljno je upisati:

```
import Image

img = Image.open ("python.png")
print img.size
img.save ("python.jpg")
```

Metoda `save` će sama, u ovisnosti o navedenoj ekstenziji, obaviti potrebnu pretvorbu i spremiti datoteku. Ukoliko se ne navede ekstenzija, spremljena datoteka će biti istog tipa kao i ulazna.

Osnovni podaci o slici se dobivaju na sljedeći način:

```
img.size #vraca tuple velicine slike
img.mode #daje informaciju o tipu slike
```

Osnovne funkcije, poput rezanja, rotiranja i sl. je moguće obaviti pozivanjem određene metode. Primjerice, rezanje slike (eng. *crop*) se svodi na definiranje okvira koji će se iz slike izrezati i pozivanjem metode `crop`:

```
okvir = (50, 50, 100, 100)
izrezani_dio = img.crop (okvir)
```

Geometrijske transformacije se također mogu jednostavno obaviti: promjena veličine slike se

obavlja pozivanjem metode `resize`, a rotiranje slike pozivanjem metode `rotate`. Metode `resize` kao argument uzima *tuple* sa dva elementa koji predstavljaju širinu i visinu, a metoda `rotate` kao argument uzima cjelobrojnu vrijednost koja govori za koliko će se stupnjeva slika rotirati u smjeru obrnutom od kazaljke na satu:

```
smanjena = img.resize ((50, 50))
rotirana = img.rotate (45)
```

3.3 SQLite

Budući da se postavio zadatak da se sve promjene napravljene nad unešenim slikama ne spremaju direktno na izvorne slike, došlo je do potrebe za korištenje baze podataka. Za ovu primjenu je korišten *SQLite3* paket koji dolazi uz instalaciju *Python*-a.

SQLite omogućava korištenje baze podataka koja ne zahtijeva zaseban poslužiteljski proces i omogućuje pristup bazi korištenjem nestandardnog *SQL* jezika. *SQLite* se također može koristiti u svrhu testiranja, prije nego se aplikacija prenese na poslužitelj.

3.3.1 Osnovne funkcije

Glavna klasa ovog modula je `Connection` i njezina instanca predstavlja bazu podataka. Primjerice, ukoliko se baza zove `proba.db`:

```
import sqlite3
connection = sqlite3.connect ("proba.db")
```

Nakon što se dobije `con` objekt, pomoću objekta `Cursor` i njegove metode `execute ()` je moguće izvršavati SQL naredbe. Primjer kreiranja tablice:

```
cursor = connection.cursor ()
cursor.execute("""CREATE TABLE Gradovi
        (
        Id integer PRIMARY KEY AUTOINCREMENT,
        Naziv TEXT
        )""")
connection.commit ()
```

Kao što je prethodno spomenuto, `connection` je objekt koji predstavlja vezu sa bazom podataka. Objekt `cursor` služi za procesiranje upita nad bazom. Primjer toga se nalazi u 2. redu gornjeg koda: u bazi stvori tablicu (`CREATE TABLE`) gradovi koji će imati stupce `ID` i

`Naziv`. Stupac `ID` je glavni ključ (`PRIMARY KEY`), što znači da će svaki podatak u tablici biti predstavljen sa elementima tog stupca. Iz toga proizlazi da glavni ključ mora biti jedinstven za svaki redak u tablici. Argument `AUTOINCREMENT` označava da će se svaki puta kada se doda novi element u stupac `Grad`, automatski će se pridružiti i vrijednost polja `ID` koja će svaki puta biti uvećana za 1. Zadnja linija poziva metodu `commit` koja sprema promjene u bazu.

Nakon što je kreirana baza, moguće je u nju staviti potrebne podatke. U primjeru ispod će se svi predmeti liste `gradovi` pohraniti u bazu. Obzirom da imena gradova mogu sadržavati znakove koji nisu unutar *ASCII* seta znakova, potrebno je svim *stringovima* koji će se unositi u bazu staviti prefiks 'u'. Na taj način se interpreteru daje do znanja da se koristi *Unicode* set znakova:

```
gradovi = [u'Zagreb', u'Karlovac', u'Osijek', u'Rijeka', u'Split']
query = "INSERT INTO Gradovi (Naziv) VALUES (?)"
for grad in gradovi:
        cursor.execute (query, [grad])
connection.commit ()
```

Novi redak se u tablicu unosi pomoću naredbe `INSERT`. Potrebno je definirati u koju tablicu se vrijednost unosi i koja vrijednost (ili vrijednosti). Kako bi se olakšao unos podataka u Python jeziku, u sami upit se ne upisuje vrijednost nego se stavlja upitnik na čije se mjesto metodom `execute` postavlja određena vrijednost. Metoda `execute` se poziva na sljedeći način:

```
cursor.execute (upit, lista_vrijednosti)
```

Čak ako se unosi samo jedan element, on opet mora biti naveden u listi.

Promjena postojećih podataka je također moguća. Primjerice, u kreiranoj tablici se želi promijeniti grad *Osijek* u *Slavonski Brod*. Prvo je potrebno saznati koji je *Id* retka u kojem se nalazi vrijednost `Osijek`. To se postiže korištenjem naredbe `UPDATE` na sljedeći način:

```
query = "UPDATE Gradovi SET Naziv = ? WHERE Naziv = ?"
cursor.execute (query, ["Osijek", "Slavonski Brod"])
connection.commit ()
```

Brisanje podataka iz baze se također obavlja po sličnom principu. Primjerice, iz baze je potrebno izbrisati vrijednost *Karlovac*:

```
query = "UPDATE Gradovi SET Naziv = ? WHERE Naziv = ?"
cursor.execute (query, [u"Osijek", u"Slavonski Brod"])
connection.commit ()
```

3.3.2 Vanjski ključ

Ukoliko se u bazi podataka želi imati, primjerice, popis pisaca i njihov grad stanovanja, nije poželjno napraviti bazu kao u prethodnim primjerima. Naime, ukoliko postoji veliki broj pisaca u bazi, velika je šansa da će mnogi od njih biti iz istih gradova. Ako se to uzme u obzir, nema smisla da se u bazi nebrojeno puta zapisuje isti podatak. Takav odnos se naziva "jedan-prema-više" (eng. *one-to-many*)" i rješava se korištenjem vanjskog ključa.

Kreiranje takve tablice bi se postiglo na sljedeći način:

```
query = """CREATE TABLE Pisci(
                Id INTEGER PRIMARY KEY AUTOINCREMENT,
                Ime_i_prezime TEXT,
                Grad INTEGER,
                FOREIGN KEY (Grad) REFERENCES Gradovi(Id))"""
cursor.execute(query)
connection.commit ()
```

Kao što se vidi, polje `Grad` više nije tipa `text` nego je sada `integer`. Izrazom `FOREIGN KEY` se daje do znanja da vrijednost u polju `Grad` odgovara `Id` polju u tablici *Gradovi*. Primjerice, unos *Miroslav Krleža, Zagreb* se može unijeti na sljedeći način:

```
ime_prezime = u"Miroslav Krleža"
grad = u"Zagreb"

query = "SELECT Id FROM Gradovi WHERE Naziv = ?"
cursor.execute (query, [grad])
id_grada = cursor.fetchone()[0]

query = "INSERT INTO Pisci (Ime_i_prezime, Grad) VALUES (?, ?)"
cursor.execute (query, [ime_prezime, id_grada])
connection.commit()
```

Srednji dio gornjeg kôda, iz prve tablice traži ID koji je vezan uz *Zagreb*. Nakon toga se jednostavno u tablicu *Pisci* unosi ime pisca i ID grada.

3.3.3 Povezivanje tablica

Tablica *Gradovi* je definirana na sljedeći način:

Tablica 3.1: Primjer - Popis gradova

ID	Ime grada
1	Zagreb
2	Slavonski Brod
3	Rijeka
4	Split

Kako bi se prikazali različiti primjeri povezivanja tablica, definirana je i tablica književnika:

Tablica 3.2: Primjer - Pisci

ID	Ime pisca	Grad rođenja
1	Miroslav Krleža	1
2	Marko Marulić	4
3	August Šenoa	1
4	Ivan Gundulić	
5	Dragutin Tadijanović	2

Valja primjetiti kako je u tablici pisaca namjerno izostavljeno Gundulićevo mjesto rođenja, i da se u tablici sa popisom gradova nalazi i grad Rijeka koji niti jedan navedeni književnik nema kao mjesto rođenja.

Unutrašnji spoj

Ključna riječ `INNER JOIN` unutar naredbe `SELECT` vraća retke kada se nađe barem jedan par koji se preklapa iz obje tablice. Takav način spajanja se naziva unutrašnji spoj (eng. *Inner join*).:

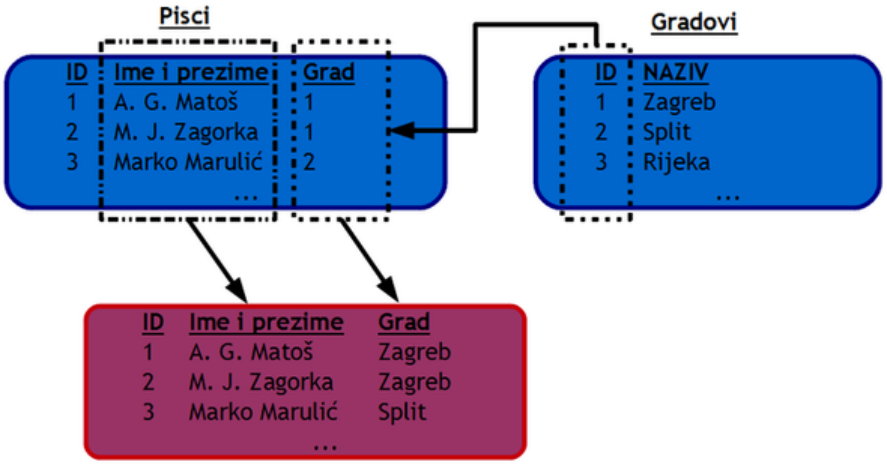
```
query = """SELECT Pisci.Ime_i_prezime, Gradovi.Naziv FROM Pisci
        INNER JOIN Gradovi
        ON Pisci.Grad=Gradovi.Id
        ORDER BY Pisci.Ime_i_prezime""" #Sortiranje po imenu pisca
cursor.execute (query)
popis = cursor.fetchall ()
```

U gornjem upitu je definirano da se tablica *Pisci* poveže sa tablicom *Gradovi* na način da se traže jednake vrijednosti stupca *Grad* u tablici *Pisci*, i stupca *Id* u tablici *Gradovi*. Naredba `ORDER BY` dobiveni rezultat sortira prema imenu pisaca. Ukoliko se upit izvrši, dobiva se rješenje prikazano tablicom ispod.

Tablica 3.3: Primjer - Unutrašnji spoj

Ime pisca	Grad rođenja
August Šenoa	Zagreb
Dragutin Tadijanović	Osijek
Marko Marulić	Split
Miroslav Krleža	Zagreb

Jedini pisac koji ovdje nije naveden je *Ivan Gundulić* jer se za *Grad rođenja* iz tablice *Pisci* za tog pisca jedino nije našao odgovarajući par u tablici *Gradovi*, stupac *Id*.



Slika 3.1: Povezivanje tablica - shematski prikaz

Direktni spoj

Direktni spoj (eng. *Cross join*) predstavlja povezivanje više tablica bez korištenja neke ključne riječi. Ovakav način spajanja se još naziva i Kartezijski produkt:

```
query = "SELECT Pisci.Ime_i_prezime, Gradovi.Naziv FROM Pisci, Gradovi"
cursor.execute (query)
popis = cursor.fetchall ()
```

Dakle, radi se o najobičnijem upitu koji će dati, za već opisani slučaj, nepoželjan rezultat.

Tablica 3.4: Primjer - Direktni spoj

Ime pisca	Grad rođenja
Marko Marulić	Zagreb
Marko Marulić	Osijek
Marko Marulić	Rijeka
Marko Marulić	Split
Ivan Gundulić	Zagreb
Ivan Gundulić	Osijek
Ivan Gundulić	Rijeka
Ivan Gundulić	Split
Dragutin Tadijanović	Zagreb
Dragutin Tadijanović	Osijek
Dragutin Tadijanović	Rijeka
Dragutin Tadijanović	Split
August Šenoa	Zagreb
August Šenoa	Osijek
August Šenoa	Rijeka
August Šenoa	Split
Miroslav Krleža	Zagreb
Miroslav Krleža	Osijek
Miroslav Krleža	Rijeka
Miroslav Krleža	Split

Kao što je vidljivo, navedenim upitom su dobivene sve moguće kombinacije povezivanja prve i druge tablice. Ukoliko se direktni spoj ograniči korištenjem uvjeta `WHERE`, dobiva se potpuno jednaki rezultat kao kod unutrašnjeg spoja:

```
query = """SELECT Pisci.Ime_i_prezime, Gradovi.Naziv FROM Pisci, Gradovi
        WHERE Gradovi.Id=Pisci.Grad"""
cursor.execute (query)
popis = cursor.fetchall ()
```

Lijevo vanjsko spajanje

Lijevo vanjsko spajanje (eng. *Left outer join*) osigurava da se svi retci sa "lijeve" strane nalaze u rezultantnom spoju.:

```
query = """SELECT Pisci.Ime_i_prezime, Gradovi.Naziv
        FROM Pisci LEFT OUTER JOIN Gradovi
        ON Pisci.Grad=Gradovi.Id ORDER BY Pisci.Ime_i_prezime"""
cursor.execute (query)
popis = cursor.fetchall ()
```

Tablica 3.5: Primjer - Lijevo vanjsko spajanje

Ime pisca	**Grad rođenja**
August Šenoa	Zagreb
Dragutin Tadijanović	Osijek
Ivan Gundulić	**None**
Marko Marulić	Split
Miroslav Krleža	Zagreb

Kao što se sada može vidjeti, za razliku od unutrašnjeg spajanja - sada se na popisu pojavio i *Ivan Gundulić* koji nema naveden grad rođenja (isti bi rezultat bio da je zadana referenca koja u drugoj tablici ne postoji). Rezultat *None* koji pored njega stoji označava da za referencu koja je bila zapisana u stupcu *Grad*, nije nađena ista vrijednost u stupcu *Id* u tablici *Gradovi*

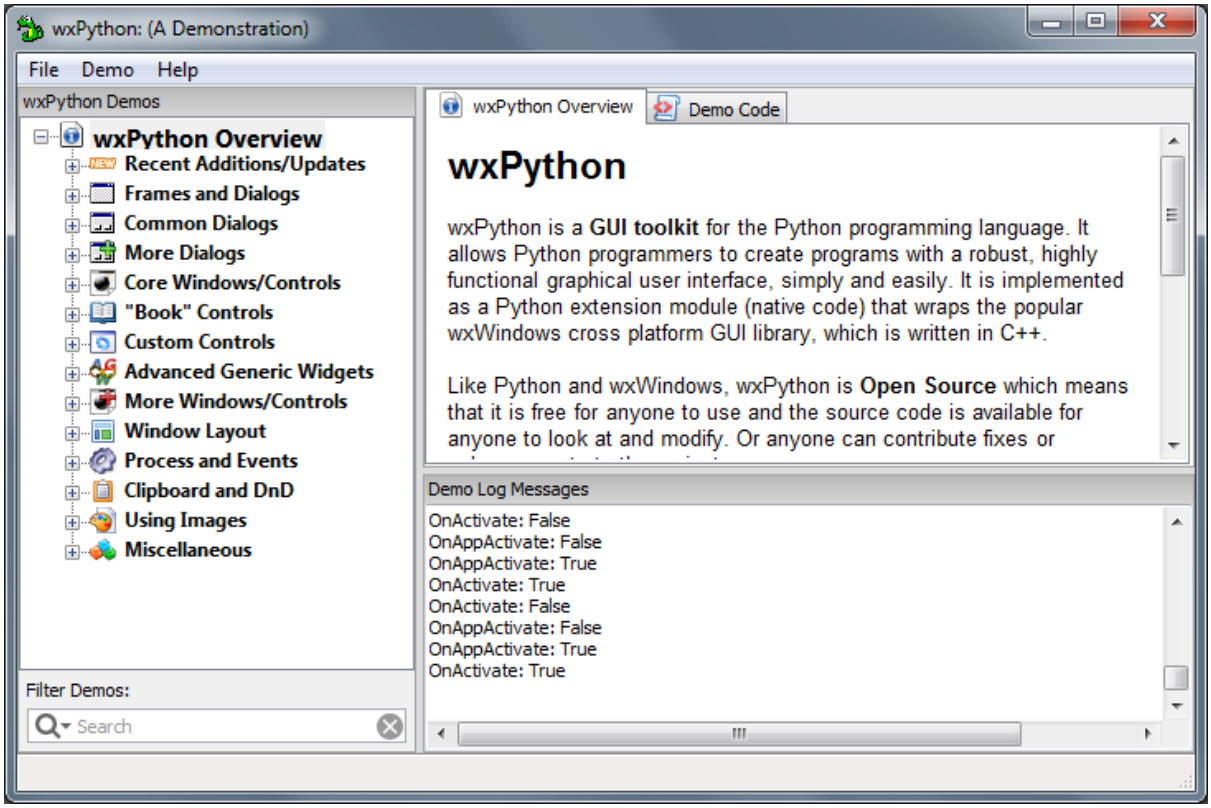
3.4 wxPython - Izrada korisničkog sučelja

Za izradu korisničkog sučelja je korišten *wxWidgets* sustav, prilagođen naravno okruženju Python-a. *WxWidgets* omogućuje korisnicima izradu programa sa robusnim i funkcionalnim grafičkim sučeljem, rad na različitim operativnim sustavima (eng. *cross-platform*), otvorenog je kôda i programima daje izgled operativnog sustava (za razliku od primjerice *tkinter* paketa).

Paket `wxPython` dolazi uz veliki broj primjera kojima se lagano mogu vidjeti njegove mogućnosti, a isto tako i vidjeti iz prve ruke kako se neki elementi mogu implementirati u vlastitu aplikaciju. Sadrži veliki broj elemenata koji se mogu dodati i uz sve to - u potpunosti je besplatan.

Na žalost, kao i sa svakim sličnim sustavom, postoje i određene loše strane:

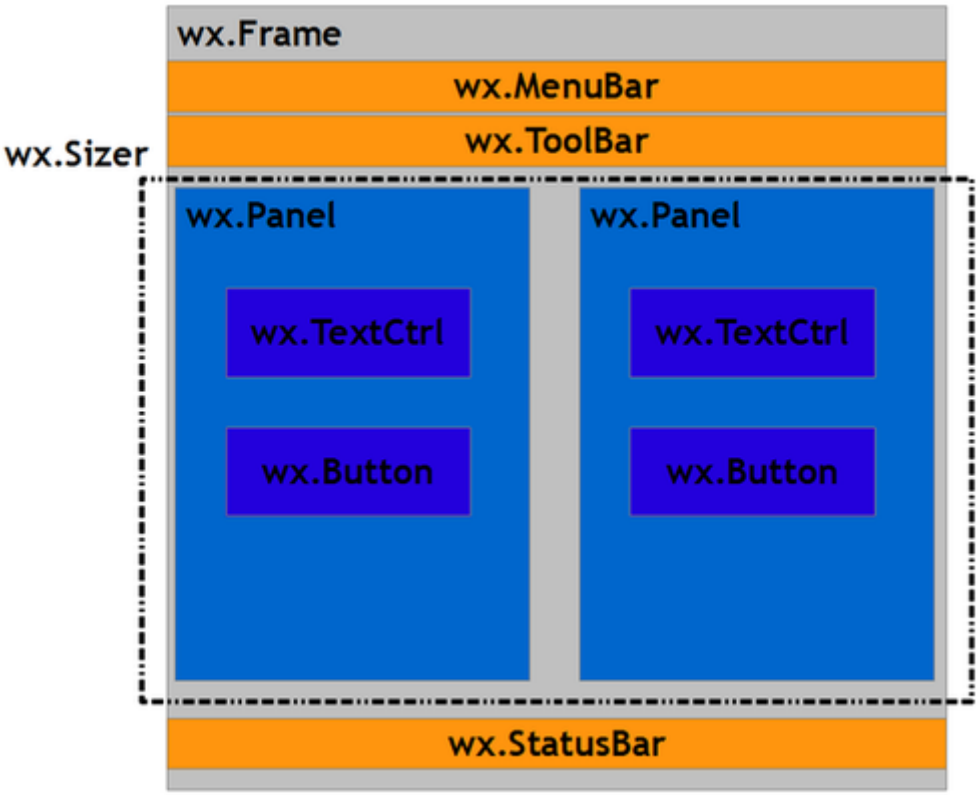
- nije podoban za komercijalnu primjenu jer mu nedostaju komponente za prikaz grafikona, te ćelije kojima se direktno prikazuje i mijenja sadržaj baze, ...
- postoji veliki broj elemenata čiji razvoj ovisi o zajednici korisnika. Elementi koji se najčešće koriste obično nemaju nikakvih *bug*-ova, ali elementi koji se rijetko koriste mogu sadržavati ponekoji, neotkriveni *bug*
- ...



Slika 3.2: wxPython - Pregled primjera

Svaka Python aplikacija koja koristi *wx* za izradu GUI-a mora imati instanciranu *wxApp* klasu. Naime, *App* je osnovna klasa u *wx* sustavu koja kreira petlju kojom se različiti događaji (eng. *event*) šalju samoj aplikaciji. Uz to, ona postavlja osnovna svojstva aplikacije i zapravo pokreće cijeli *wx* alat. Svi elementi grafičkog sučelja bi se trebali kreirati nakon instanciranja *wxApp* klase.

Jednom kada se osigurala glavna petlja kojom će se izmjenjivati događaji, može se pristupiti postavljanju elemenata grafičkog sučelja. Glavna klasa iz koje se deriviraju sve ostale je *wxWindow* i ona predstavlja sve vidljive objekte na zaslonu. Prva klasa po hijerarhiji koja je i vidljiva na zaslonu je *wxFrame*. *wxFrame* je glavni objekt u koji se dodaju ostali elementi grafičkog sučelja. Ti elementi se mogu podijeliti u nekoliko skupina (naravno, na više načina) bitnih za izradu bilo koje aplikacije. To su: glavni prozor na koji se postavljaju paneli na koje se dalje postavljaju elementi za upravljanje (liste, gumbovi i sl.). Još jedna skupina bi bili elementi koji se dodaju glavnom prozoru, ali može postojati samo jedna njihova instanca. To je primjerice alatna ili statusna traka (eng. *toolbar* i *statusbar*).



Slika 3.3: Hijerarhija *wx* elemenata

3.4.1 Glavni elementi korisničkog sučelja

wx.App

Obzirom da svaka aplikacija mora imati instancu *wxApp* klase, glavna petlja aplikacije se definira na sljedeći način:

```
app = wx.App()
view = View (app, "Test")
app.MainLoop()
```

U gornjem primjeru, `app` je instanca klase `wx.App`, `view` je instanca klase `View` koja će se kasnije opisati. `app.MainLoop()` poziva metodu `MainLoop` objekta `app`. odnosno započinje glavnu petlju.

wx.Frame

Kao što je već spomenuto, *wxFrame* je glavni vidljivi objekt i u njega se dodaju svi ostali elementi.:

```
class View(wx.Frame):
        def __init__(self, parent, title):
                wx.Frame.__init__(self, wx.GetApp().TopWindow, title=title,
                style=wx.DEFAULT_FRAME_STYLE)

                self.Show()
```

Svi elementi koji su niži po hijerarhiji, nasljeđuju metode klasa (elemenata) iznad sebe. Obzirom da su sve niže klase derivirane iz `wx.App` klase, nužno je u svakoj imati `__init__` funkciju. U gornjem primjeru se definira klasa *View* koja nasljeđuje sve metode klase `wx.Frame`. Iz tog razloga nije potrebno imati instancu `wx.Window` klase, jer će sve njene metode biti unutar klase `wx.Frame`. U `__init__` dijelu su argumenti: koja se klasa inicijalizira (`self`), koji je njezin *parent* element i koji je naslov (argument `title`). Nakon toga se sa tim argumentima provodi inicijalizacija `wx.Frame` klase, s time da se navode neki argumenti za koje se želi da budu uvijek jednaki (npr. `style`). U gornjem primjeru je navedeno da će izgled prozora biti neki predefinirani.

U zadnjoj liniji kôda gornjeg primjera se poziva metoda `Show` koja ne radi ništa drugo nego prikazuje glavni prozor na ekranu, skupa sa svim elementima. Ova se metoda može pozivati nad svim elementima. Naime, svi se elementi prikazuju skupa sa glavnim prozorom. Oni koji se naknadno dodaju se također prikazuju. No, metoda suprotna metodi `Show` je `Hide` koja sakriva element nad kojim je pozvana. Iz tog razloga je potrebna metoda `Show` kako bi se ponovno prikazali sakriveni elementi.

wx.Panel

Jednom kada je definiran glavni prozor aplikacije, potrebno je staviti elemente na koje se mogu postaviti elementi za upravljanje. Ovo se može zamisliti kao pano za obavijesti. Primjerice, postoji pano (*Frame*), na koji se stavljaju različiti papiri (*Panel*) koji na sebi ne moraju imati ništa ili mogu mogu imati neki tekst, sliku i sl ...

`wxPanel` se definira vrlo jednostavno unutar `__init__` ili neke druge metode

```
Button1 = wx.Button (Panel, id=1, label="Uvecaj")
Button2 = wx.Button (Panel, id=2, label="Umanji")
Prikaz_teksta = wx.TextCtrl (Panel, id=-1, value="1",
                    style=wx.TE_READONLY)
```

Gornjim primjerom se definiranom *Frame*-u dodao `wx.Panel` objekt. Jedini navedeni argument je `self` čime se daje do znanja da element pripada `Frame` elementu. Nakon toga su *panelu* dodana dva gumba i polje za prikaz teksta, čiji će način korištenja kasnije biti detaljnije opisan.

wx.StaticBox

`StaticBox` je ništa drugo nego pravokutnik oslikan oko neke skupine elemenata. Na taj način ih se može vizualno odvajati po funkciji ili nekom drugom logičkom razmještaju. Primjer je dan kasnije prilikom opisivanja elementa `StaticBoxSizer`.

wx.ToolBar

Gotovo svaka aplikacija ima alatnu traku pomoću koje se brzo pristupa najčešćim radnjama. Definira se kao i prethodni elementi:

```
toolbar = self.CreateToolBar()
toolbar.AddLabelTool(wx.ID_OPEN, '', wx.Bitmap('Open.png'))
toolbar.Realize()
```

Prva linija kôda kreira instancu klase `CreateToolBar`. Drugom linijom kôda se dodaje novi predmet u alatnu traku. Taj predmet će prilikom pozivanja neke funkcije (opisano kasnije) pokazivati identitet (`id`) `ID_OPEN`, a ikona koja će se prikazivati će biti iz vanjske datoteke `T_Open.png`. Slično kao sa metodom `Show` kod klase `Frame`, alatna traka se ne prikazuje dok se ne pozove metoda `Realize`.

wx.MenuBar

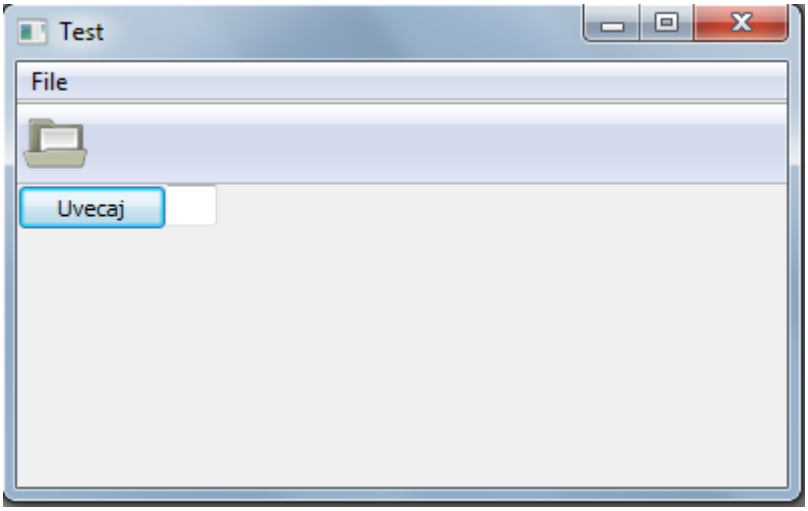
Klasa `MenuBar` služi za kreiranje glavnog izbornika programa:

```
menubar = wx.MenuBar()
datoteka = wx.Menu()
datoteka.Append(wx.ID_EXIT, 'I&zadi', 'Izlazak iz programa')
menubar.Append(datoteka, '&File')
self.SetMenuBar(menubar)
```

Glavni izbornik se definira instanciranjem klase `MenuBar`. Pojedini padajući izbornici se definiraju klasom `Menu`. Kao što se u prethodnom primjeru vidi, prvo se klasa `MenuBar` instancira u objekt `menubar`, zatim se definira padajući izbornik `datoteka` kojem se dodaje predmet za izlazak iz programa. Nakon toga se glavnom izborniku dodaje taj padajući izbornik, a sami glavni izbornik se pridružuje svome *parent* objektu.

3.4.2 Nevidljivi elementi grafičkog sučelja

Ako se cijeli prethodno navedeni kôd poveže u jednu cjelinu i pokrene, dobiva se izgled grafičkog korisničkog sučelja prikazan na slici ispod.



Slika 3.4: Jednostavno grafičko korisničko sučelje

Kao što se sa slike i može vidjeti, gumbi koji su se postavili se misteriozno nalaze na alatnoj traci. Teško da je to raspored elemenata koji bi netko želio. Iz tog razloga je nužno koristiti elemente koji služe upravo za definiranje željenog rasporeda elemenata. Ti se elementi nazivaju *Sizer*-i.

Da bi se *Sizer* definirao, prvo mu je potrebno odrediti tip, *parent* element i sve elemente koji ulaze u njega i na koji način (horizontalno i/ili vertikalno slaganje elementa po element). *Sizer* će svoju veličinu prilagoditi elementima unutar sebe i zatim se njegov *parent* element može prilagoditi veličini *sizera*. Na ovaj način se izbjegava besmisleno slaganje elemenata kao u prethodnom primjeru.

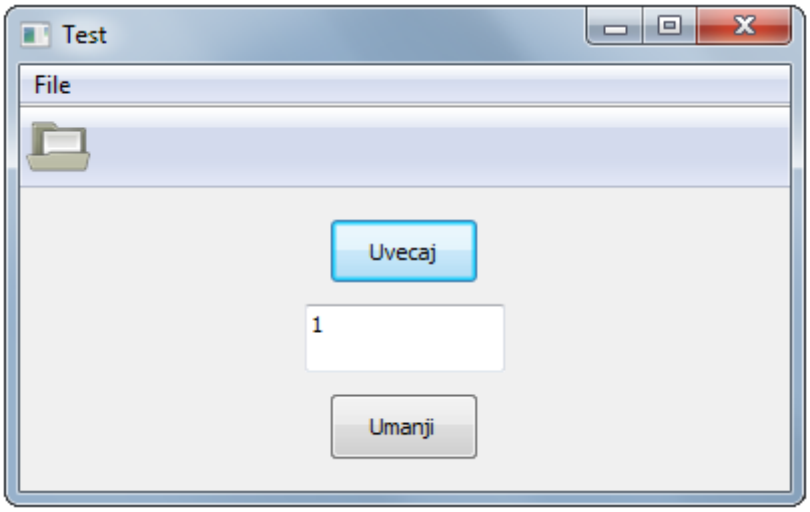
wxBoxSizer

`BoxSizer` slaže dodane (*child*) elemente ili horizontalno ili vertikalno, ovisno o tome kako se definira. Prilikom dodavanja elemenata, može se odrediti i njihovo poravnanje u odnosu na sami *sizer*.:

```
Box = wx.StaticBox (Panel, -1, "Naslov") #
Sizer = wx.StaticBoxSizer (Box, wx.VERTICAL)
Sizer.Add ((0,15))
Sizer.Add (Button1, 1, wx.ALIGN_CENTRE_HORIZONTAL)
Sizer.Add ((0,10))
Sizer.Add (Prikaz_teksta, 1, wx.ALIGN_CENTER_HORIZONTAL)
Sizer.Add ((0,10))
Sizer.Add (Button2, 1, wx.ALIGN_CENTRE_HORIZONTAL)
Sizer.Add ((0,15))
Panel.SetSizerAndFit (Sizer)
```

Gornji primjer definira `BoxSizer` koji će elemente slagati vertikalno. Nakon toga mu se

dodaje prvi gumb. Argumenti unutar metode `Add` su sljedeći: naziv elementa, faktor rastezanja - koliko se element smije rastegnuti u odnosu na svoju inicijalnu (minimalnu) veličinu, te zadnji argument predstavlja tip poravnanja u odnosu na `Sizer`. Nakon toga se dodaje praznina širine 0 i visine 100 piksela, zatim drugi gumb. Nakon toga slijedi postavljane *Sizera*. Obzirom da je *panel* jedini direktni *child* element glavnog prozora, pomoću sizera će se posložiti elementi na *panelu* i veličina *panela*.



Slika 3.5: Jednostavno grafičko korisničko sučelje

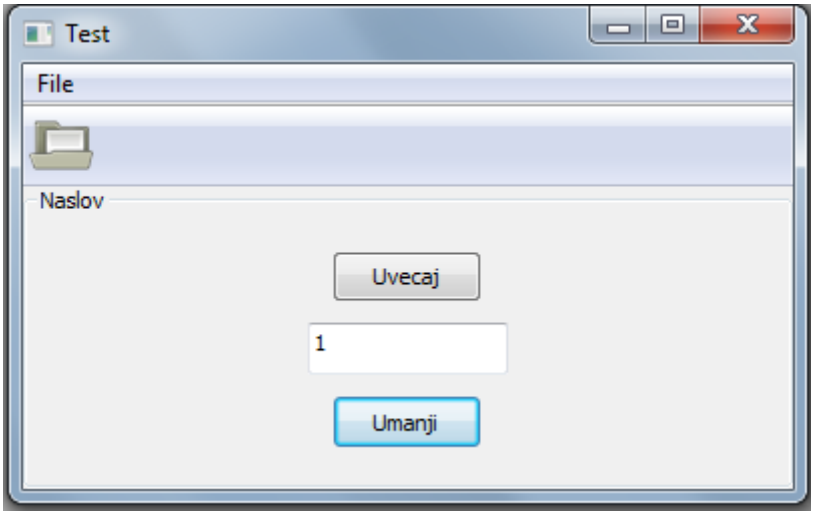
wxStaticBoxSizer

Ovaj *sizer* je funkcionalno potpuno jednak *BoxSizer*-u. Jedina razlika je u tome što ovaj *sizer* koristi element `wxStaticBox` na koji postavlja ostale elemente. Da bi se koristio, kôdu iz prethodnog primjera je umjesto prve linije kôda potrebno unijeti sljedeći kôd:

```
Box = wx.StaticBox (Panel, -1, "Naslov")
Sizer = wx.StaticBoxSizer (Box, wx.VERTICAL)
```

wxGridSizer i wxFlexGridSizer

`GridSizer` je u neku ruku sličan `BoxSizer`-u. Razlika je u tome što `GridSizer` raspoređuje elemente u retke i stupce, odnosno u mrežu sa jednakom širinom stupaca. Za razliku od njega, `FlexGridSizer` je fleksibilniji na način da stupci ne moraju biti jednake širine.



Slika 3.6: Jednostavno grafičko korisničko sučelje

3.4.3 Elementi za upravljanje

Osim klase `wx.Button` koji je djelomično opisan u prethodnom dijelu, postoji cijeli niz elemenata kojima se postiže interakcija između korisnika i programa. Panel, slike i sl. su statični elementi na koje korisnik ne može direktno utjecati. No, zato postoji cijeli niz elemenata pomoću kojih se šalju događaji (eng. *events*) programu koji ih po potrebi obradi i na temelju njih obavi neku funkciju.

U sljedećih nekoliko redova će biti opisani osnovni elementi za upravljanje koji su korišteni tijekom izrade rada.

wx.Button

Gumb je osnovni i najčešći element za upravljanje. Može se postaviti na gotovo bilo koji prozor (dijalog, panel, ...):

```
wx.Button (parent, id, naslov, pozicija, veličina, stil)
```

Argument *parent* označava kojem elementu će pripadati, *id* je identifikacijski broj gumba (postavlja se na -1 ako nije bitno), *naslov* je string koji će pisati na gumbu, *pozicija* predstavlja doslovce poziciju na *parent* elementu, *veličina* je tuple koji predstavlja veličinu gumba. Zadnji argument je *stil*:

- `wx.BU_LEFT`, `wx.BU_TOP`, `wx.BU_RIGHT`, `wx.BU_BOTTOM` - željeno poravnanje teksta gumba
- `wx.BU_EXACTFIT` - ne koristi standardnu veličinu gumba, nego veličinu gumba sve na minimalnu (onu koja je dovoljna da u cijelosti prikaže naslov gumba)

Događaj koji se može pratiti na ovom elementu je sljedeći:

- `EVT_BUTTON` - šalje se kada se gumb stisne

wxTextCtrl (Text Control)

Ovaj element omogućuje prikaz i unos teksta, a tekst u njemu može biti dužine jedne ili više linija

```
wx.TextCtrl (parent, id, vrijednost, pozicija, veličina, stil)
```

Svi argumenti su istovjetni prethodnom elementu, razlika je samo u argumentu *vrijednost* koja ovdje predstavlja početni zapis.

Važniji stilovi koji se mogu koristiti su:

- `wx.TE_MULTILINE` - tekst u više redova
- `wx.TE_PASSWORD` - polje za lozinku, svi znakovi će biti zamijenjeni zvjezdicama
- `wx.TE_READONLY` - prikazani tekst neće biti moguće editirati

Važniji događaji koji se mogu pratiti:

- `wx.EVT_TEXT` - javlja se svaki put kada se tekst promijeni, bilo od strane korisnika ili samog programa

wxComboBox

Ovaj element je, ovisno o stilu, padajući ili statični izbornik, a definira se na sljedeći način:

```
wx.ComboBox (parent, id, pozicija, veličina, odabiri, stil)
```

Prva dva argumenta su već opisana u prethodnim elementima, *pozicija* označava poziciju na *parent* elementu, *veličina* je doslovce veličina izbornika, *odabiri* je lista mogućih odabira padajućeg izbornika, a stil može biti jedan od sljedećih:

- `wx.CB_SIMPLE` - statični izbornik (padajući izbornik koji je cijelo vrijeme otvoren)
- `wx.CB_DROPDOWN` - padajući izbornik
- `wx.CB_READONLY` - onemogućava (čak i direktno iz kôda) odabire koji nisu na listi odabira
- `wx.CB_SORT` - sortira listu odabira po abecedi

Primjerice, ukoliko se želi postaviti padajući izbornik sa listom boja po abecedi, trebao bi se koristiti sljedeći dio kôda:

```
lista_boja = ["Crvena", "Plava", "Zelena", "Bijela", "Zuta"]
wx.ComboBox (self, -1, (0,0), (100,10), lista_boja,
             wx.CB_DROPDOWN|wx.CB_SORT)
```

Događaji koji se mogu vezati uz ovaj element su sljedeći:

- `EVT_COMBOBOX` - šalje se kada se odabere neki predmet u izborniku. Točan odabir se može dobiti na način da se objektu koji se šalje događajem, pozove metoda GetValue (npr. `evt.GetValue()`)
- `EVT_TEXT` - šalje se kada se tekst unutar izbornika promijeni. Ovo se može dogoditi samo ukoliko za stil nije definiran `wx.CB_READONLY`
- `EVT_TEXT_ENTER` - šalje se kada se u izborniku pritisne tipka `Enter`

3.4.4 Dijalozi

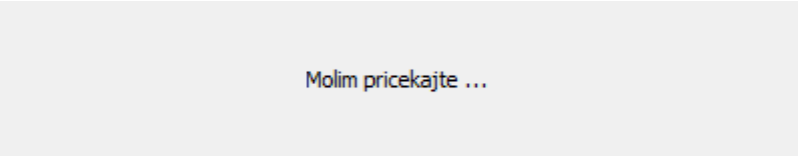
WxPython sadrži veliki broj dijaloga kojima se olakšava odabiranje/prikaz datoteke, prijava greške korisniku i sl.

Dijalog zauzeća

Ovaj dijalog je vrlo koristan kako bi se korisniku dalo do znanja da program nešto radi. Primjerice:

```
busy = wx.BusyInfo ("Molim pricekajte ...")
neka_funkcija_koja_traje_dugo
busy = None
```

U trenutku kada se instancira klasa `wx.BusyInfo`, pojaviti će se dijalog i biti će prikazan sve dok se pripadajućem objektu ne pridruži vrijednost `None`.



Slika 3.7: Dijalog zauzeća

Info dijalog

Info dijalog je najjednostavniji dijalog koji služi za slanje neke informacije korisniku:

```
wx.MessageBox('Prikazana poruka', 'Naslov',
              style=wx.OK | wx.ICON_INFORMATION)
```

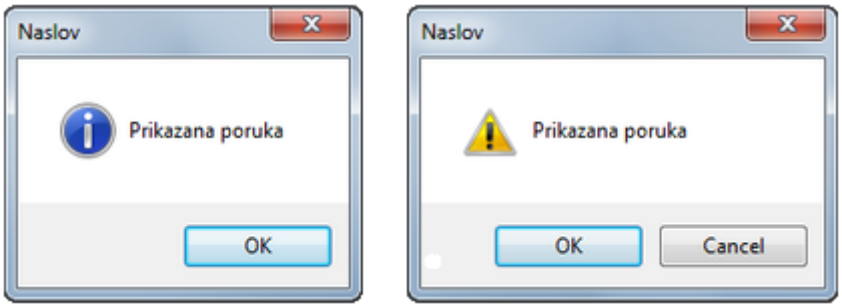
Prva dva potrebna argumenta su jasna, oni predstavljaju sadržaj poruke i naslov dijaloga. Posljednji argument omogućava prilagodbu. U gornjem primjeru će dijalog sadržavati gumb OK i pored poruke će biti stavljena ikona koja označava informaciju.

Tablica 3.6: Mogući argumenti za stil dijaloga

Argument	Opis
wx.OK	prikazuje gumb OK
wx.CANCEL	prikazuje gumb CANCEL
wx.YES_NO	prikazuje YES i NO gumbe
wx.YES_DEFAULT	gumb YES je predefinirani odabir
wx.NO_DEFAULT	gumb NO je predefinirani odabir
wx.ICON_EXCLAMATION	Prikazuje ikonu za upozorenje
wx.ICON_ERROR	Prikazuje ikonu za pogrešku
wx.ICON_HAND	Ista kao i wx.ICON_ERROR
wx.ICON_INFORMATION	Prikazuje ikonu za informaciju
wx.ICON_QUESTION	Prikazuje ikonu za upit”

Vidljivo je da je dijalog moguće prilagoditi vlastitim potrebama, primjerice ukoliko želimo gumbe OK i CANCEL, te da dijalog označava neko upozorenje:

```
wx.MessageBox('Prikazana poruka', 'Naslov',
              wx.OK | wx.CANCEL | wx.ICON_WARNING)
```



Slika 3.8: Primjer info dijaloga

Dijalog za otvaranje datoteke

Ovim dijalogom se omogućuje jednostavno otvaranje jedne ili više datoteka:

```
dlg = wx.FileDialog (self, title="Odaberi datoteku",
                     style=wx.OPEN|wx.MULTIPLE)
if dlg.ShowModal() == wx.ID_OK:
        print dlg.GetPaths ()
dlg.Destroy ()
```

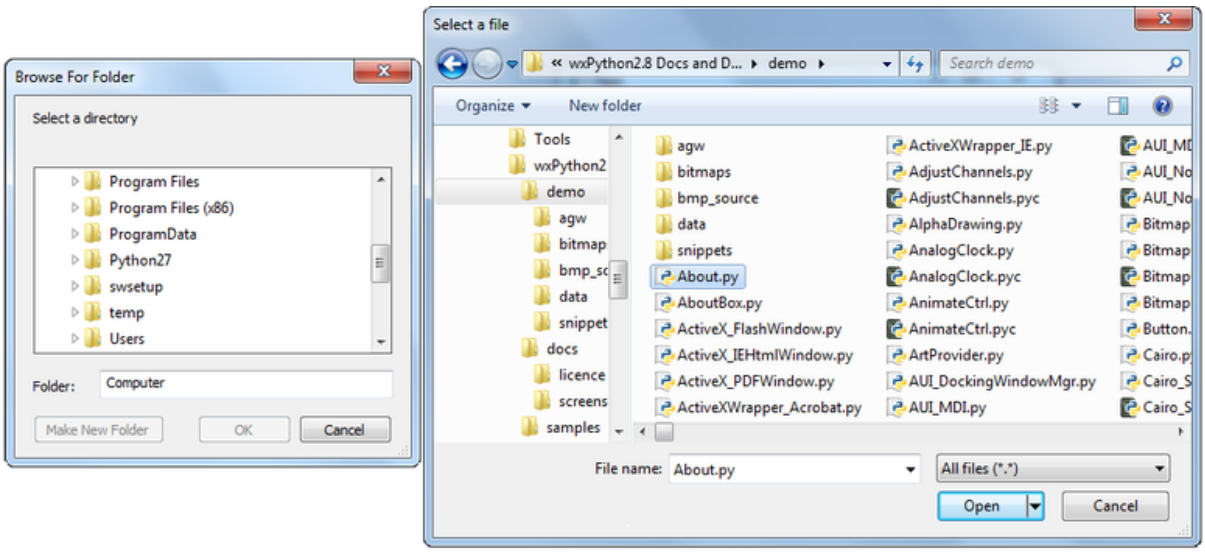
Izvršavanjem gornjeg primjera će se pojaviti dijalog u kojem se može odabrati jedna ili više datoteki. Odabirom gumba *OK* će se ispisati putanje odabranih datoteki. Ukoliko se želi izbor datoteki ograničiti samo na jednu, iz argumenta `style` treba izbaciti zastavicu `wx.MULTIPLE`.

Dijalog za odabir direktorija

Ovim dijalogom se jednostavno odabire neki direktorij:

```
dirdlg = wx.DirDialog (self, "Odaberi direktorij:")
if dirdlg.ShowModal () == wx.ID_OK:
        print dirdlg.GetPath ()
dirdlg.Destroy ()
```

Gornjim primjerom se dobiva dijalog za odabir direktorija i u kada se neki odabere i stisne gumb `OK`, ispisuje se odabrani direktorij. Posljednjom linijom kôda se dijalog briše.



Slika 3.9: Odabir direktorija (lijevo) i odabir datoteke (desno)

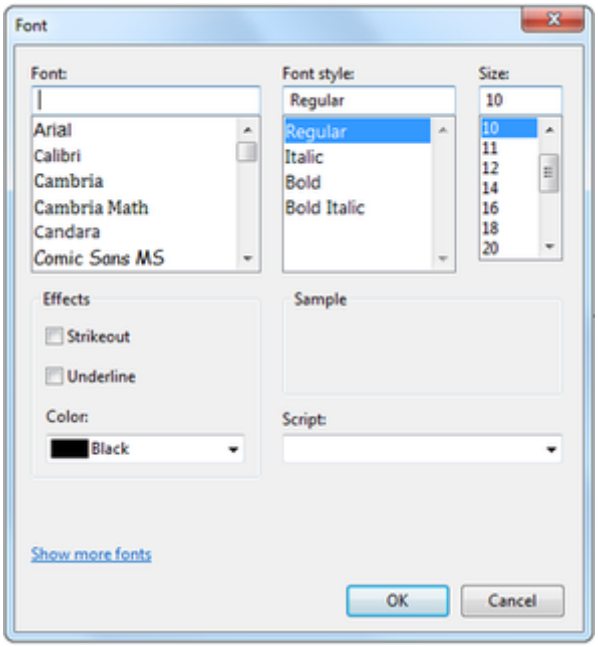
Odabir fonta

U gotovo svakoj aplikaciji je potrebno korisniku dati mogućnost da si prilagodi sučelje odabirom *Fonta*. Kako bi se to olakšalo, postoji već gotovi dijalog pomoću kojeg se korisniku daje mogućnost odabira:

```
data = wx.FontData()

dlg = wx.FontDialog(self, data)
if dlg.ShowModal() == wx.ID_OK:
    data = dlg.GetFontData()
    print data.GetChosenFont()
    print data.GetColour()
```

Klasa `FontData` sadrži sve potrebne metode za odabir fonta. Njezin glavni zadatak je "komunikacija" sa dijalogom za izbor fonta. Naime, inicijalizacija dijaloga za izbor fonta zahtijeva objekt tipa `FontData`. Početne vrijednosti dijaloga će biti jednake onima definiranim u objektu `data`. Jednom kada se stisne gumb `OK`, pomoću metode `GetFontData` će se izabrane vrijednosti upisati u objekt `data`. Kasnije se taj objekt može koristiti prilikom izbora fonta i boje, kao što je navedeno u zadnje dvije linije kôda.



Slika 3.10: Odabir fonta

3.4.5 Dodavanje funkcija elementima putem događaja

Pod pojmom događaji (eng. *event*) se doslovce smatraju svi događaji koje korisnik u aplikaciji napravi. To može biti pomicanje miša ili unos putem tipkovnice, ali može biti primjerice i promjena veličine prozora ili gašenje prozora. Osnovni pojmovi koje treba razlikovati su:

- *Event loop* (petlja događaja) - u svakoj *wx* aplikaciji je to već opisana metoda `MainLoop()`
- *Event handler* (rukovatelj događaja) - metode koje reagiraju na određeni događaj
- *Event type* (vrsta događaja) - klik na gumb, minimiziranje prozora, odabir u izborniku, ...
- *Event binder* (poveznik događaja) - povezuje određeni tip događaja sa rukovateljem događaja

Obzirom da su elementi za upravljanje potpuno beskorisni ukoliko se na njih ne veže neka funkcija. Metoda `Bind` je poveznik događaja koji se može koristiti na gotovo svakom elementu sučelja:

naziv_elementa.Bind (dogadaj, funkcija)

Primjerice, na sučelju se nalazi samo gumb kreativnog naziva `Gumb`. Naslov tog gumba je `1` i želim se da svaki put kada se gumb klikne, taj broj uveća za 1. Prvo je nužno napraviti gumb i pridružiti mu metodu za određeni događaj(sve se radi unutar glavne petlje programa):

```
Gumb = wx.Button (Panel, id=1, label="1")
Gumb.Bind (wx.EVT_BUTTON, self.Akcija)
```

Funkcija koja se poziva je definirana na sljedeći način:

```
def Akcija (self, event):
    element = event.GetEventObject ()
    vrijednost = element.GetLabel ()
    nova_vrijednost = eval (vrijednost)+1
    element.SetLabel (str(nova_vrijednost))
```

Svaki put kada se klikne na gumb, pozvati će se funkcija `Akcija`. Toj će se funkciji automatski proslijediti argument `event` koji sadrži sve informacije o događaju: tip događaja, ID elementa nad kojim je pozvan, ... Tako će se u gornjem primjeru iz argumenta `event` metodom `GetEventObject` doći do objekta nad kojim je došlo do događaja. Taj se podatak zapisuje u varijablu `element` (ona je zapravo pokazivač na objekt gumba). Nakon toga se uzima naslov gumba, i njegovoj brojčanoj vrijednosti se dodaje 1, te se zatim nova vrijednost postavlja kao naslov gumba.

U prethodnim primjerima izrade korisničkog sučelja su bila 2 gumba i jedno polje za prikaz teksta. Ukoliko se ne želi svakom zasebnom elementu pridruživati zasebna funkcija, moguće je `event` argument iskoristiti na drugačiji način. Primjerice, potrebno je napraviti funkciju koja će pritiskom na gumb *Uvećaj*, uvećati vrijednost u tekstualnom polju za 1, a pritiskom na gumb *Umanji* će se ta ista vrijednost umanjiti za 1.:

```
def Akcije (self, event):
        id_gumba = event.GetId ()
        element = event.GetEventObject ()
        vrijednost = self.Prikaz_teksta.GetValue ()

        if id_gumba == 1:
                nova_vrijednost = eval (vrijednost)+1
        elif id_gumba == 2:
                nova_vrijednost = eval (vrijednost)-1
        else:
                raise "Nepoznati ID"

        self.Prikaz_teksta.SetValue (str(nova_vrijednost))
```

Funkcije vezane uz događaje nije nužno definirati samo za elemente za upravljanje - primjerice, ukoliko se u glavnom prozoru želi saznati na kojoj je koordinati prozora korisnik napravio desni klik mišem. U tom slučaju bi se glavnom prozoru pridružila funkcija koja bi bila definirana na sljedeći način:

```
def ProzorXY (self, event):
        x = event.GetX()
        y = event.GetY()
        print x,y
```

3.4.6 Crtanje

Klasa `wx.DC` omogućava crtanje po određenim elementima grafičkog sučelja (uglavnom po `wx.Panel` objektima). Na taj način se vrlo jednostavno mogu nacrtati različiti geometrijski likovi, dodati druge slike, ... Svaki put kada se elementima promijeni veličina, dolazi do događaja `EVT_PAINT`. Obzirom da sve što je nacrtano, nije nacrtano na samoj slici nego samo "lebdi" iznad elementa - potrebno ga je prilikom svakog osvježavanja prozora ponovno nacrtati. No, to ne predstavlja neki veliki problem, obzirom da se i vrlo složeni crteži vrlo brzo nacrtaju (100 ispunjenih poligona za pola sekunde na prosječnom računalu iz 2008. godine). Primjerice, na praznome `wx.Panel` objektu je potrebno nacrtati pravokutnik:

```
import wx

class Panel (wx.Panel):
        def __init__(self, parent, id):
                wx.Panel.__init__(self, parent, id)
                self.SetBackgroundColour("white")
                self.Bind(wx.EVT_PAINT, self.OnPaint)

        def OnPaint(self, evt):
                self.dc = wx.PaintDC(self)
                self.dc.BeginDrawing()
                self.dc.DrawRectangle(250,250,50, 50)
                self.dc.EndDrawing()
                del self.dc

app = wx.PySimpleApp()
frame = wx.Frame(None, -1,size=(400,400))
Panel(frame,-1)
frame.Show(True)
app.MainLoop()
```

Kao što se iz primjera vidi, događaju (`EVT_PAINT`) je pridružena funkcija `OnPaint` u kojoj se nalazi instanca osnovne klase za crtanje - `wx.PaintDC`. Nakon toga je samo preostalo pozivanje metode koja će nacrtati pravokutnik. Problem ovakvog pristupa je što kada bi imali jako veliki broj elemenata, prilikom svake promjene veličine prozora bi došlo do treperenja slike, jer se elementi ne bi mogli nacrtati u realnom vremenu.

Još jedan problem gornjeg pristupa je što se nakon crtanja ne zna gdje je koji element nacrtan, niti koji dio elementa on zapravo zauzima. Ukoliko se sa nacrtanim objektima želi manipulirati (pomicati, brisati, ...) pomoću miša, nužno je znati njihovo područje interesa (eng. *region of interest* - *ROI*) - sve točke na elementu na kojima se nalaze i točke nacrtanog objekta. Klasa `wx.PseudoDC` u sebe sprema sve nacrtane elemente, i crta ih na zadani element. Prednost te klase je ta što se u svakom trenutku zna područje interesa svakog elementa. To znači da je dovoljno odabrati neku koordinatu, a određena metoda objekta `wx.PseudoDC` klase će vratiti koji se svi elementi na toj lokaciji nalaze. Također, dodani elementi se iz objekta mogu brisati i/ili translatirati.

Za razliku od `wx.PaintDC` klase, `wx.PseudoDC` klasa se instancira van metode koja reagira na `EVT_PAINT` događaj. Metode koje su nužne za korištenje su:

- `BeginDrawing` - označava početak crtanja/dodavanja elemenata
- `EndDrawing` - označava kraj crtanja/dodavanja elemenata
- `SetId (id)` - postavlja identifikacijski broj za sve elemente dodane između početka i kraja crtanja

- `DrawBitmap (bmp, x, y, alpha)` - na (x,y) koordinati dodaje `bmp` objekt. Argumentom `alpha` se daje do znanja ima li `bmp` objekt *Alpha* kanal
- `SetIdBounds (id, rectangle)` - elementima sa identifikacijskim brojem `id` definita područje interesa kao pravokutnik određen argumentom `rectangle`

Metoda koja se obično koristi unutar metode koja reagira na `EVT_PAINT` događaj je `DrawToDCClipped (dc, r)`. Ta metoda služi da bi se `wx.PaintDC` objektu prenesu svi elementi koje treba nacrtati, ali samo u dijelu prozora koji je određen pravokutnikom `r`. Vidljivo je da je ovakav način crtanja puno brži jer nije potrebno osvježavati cijeli prikaz, nego samo određeni dio što je posebno korisno prilikom pomicanja ili brisanja dodanih objekata.

3.4.7 MVC uzorak i wxPython

U praksi svaki dio MVC arhitekture bi trebao biti nezavisan od ostalih. Model tako ne bi smio sadržavati kôd vezan uz prikaz ili upravljanje prikazom, dakle nikakav prikaz i nikakva interakcija sa korisnikom. View/Controller ne bi smio sadržavati dijelove za bilo kakav upit prema bazi - grafičko korisničko sučelje služi isključivo za prikaz.

Kako bi se olakšala "komunikacija" između pojedinih dijelova, potrebno je napraviti sustav notifikacija između između svih dijelova i to na način da pojedini dijelovi uopće ne trebaju "znati" da ostali dijelovi postoje. Srećom, to nije potrebno vlastoručno programirati jer unutar *wx* biblioteke postoji biblioteka `wx.lib.pubsub` koja služi baš za takvu komunikaciju. Primjerice, neka se uzme u obzir aplikacija koja prikazuje stanje na tekucem racunu. Za isti račun mogu postojati dvije kartice. Ukoliko osoba A i osoba B idu u isto vrijeme nešto plaćati, potrebno je pravovremeno u bazi promijeniti trenutno stanje na računu. Dakle, iznos se mora odbiti odmah nakon plaćanja. Korištenjem klase `Publisher` se dobiva sustav notifikacija, kojim se iz bilo kojeg dijela aplikacije može istovremeno obavijestiti ostale (zainteresirane) dijelove o nastaloj promjeni. Na ovaj način se spomenuta aplikacija može vrlo jednostavno definirati:

```
import wx
from wx.lib.pubsub import Publisher as pub

class View (wx.Frame):
    def __init__ (self):
        wx.Frame.__init__ (self, parent=None, title =
                           "Odbijanje iznosa sa racuna")

        sizer = wx.BoxSizer (wx.VERTICAL)
        self.trenutno_stanje = wx.TextCtrl (self, style=wx.TE_READONLY)
        self.za_odbiti = wx.TextCtrl (self)
        self.izvrsi = wx.Button (self, label="Umanji racun")
        sizer.Add (self.trenutno_stanje, -1, wx.ALIGN_CENTER|wx.EXPAND)
```

```
        sizer.Add (self.za_odbiti, -1, wx.ALIGN_CENTER|wx.EXPAND)
        sizer.Add (self.izvrsi, -1, wx.ALIGN_CENTER)
        self.SetSizer (sizer)

class Model:
    def __init__ (self):
        #dohvati stanje na racunu, primjerice 100kn je
        self.stanje = 100

    def OdbijSaRacuna (self, iznos):
        self.stanje -= iznos
        #napravi i promjenu u bazi
        pub.sendMessage ("PROMJENA STANJA RACUNA", str(self.stanje))

class Controller:
    def __init__ (self):
        self.model = Model ()
        self.view = View ()
        self.view.izvrsi.Bind (wx.EVT_BUTTON, self.OdbijIznos)

        #prikazi trenutno stanje racuna
        self.view.trenutno_stanje.SetValue("Trenutno stanje: "+ str(self.model.s

        pub.subscribe (self.Promjena_stanja, "PROMJENA STANJA RACUNA")
        self.view.Show ()

    def OdbijIznos (self, event):
        #pretpostavka je da se u polje prozora unosi
        #samo numericka vrijednost
        iznos = int(self.view.za_odbiti.GetValue ())
        self.model.OdbijSaRacuna (iznos)

    def Promjena_stanja (self, poruka):
        self.view.trenutno_stanje.SetValue ("Trenutno stanje: "+poruka.data)

app = wx.App ()
controller = Controller ()
app.MainLoop ()
```

Gore navedeni primjer daje vrlo jednostavno sučelje kojime se račun umanjuje za određeni iznos novaca. Klasa `View` sadrži isključivo elemente korisničkog sučelja. U klasi `Model`, u metodi `OdbijSaRacuna` se poziva funkcija `pub.sendMessage`. U tom trenutku će svi elementi koji su se pretplatili na obavijest u njoj, izvršiti pripadajuću metodu kojoj će se proslijediti tip i sadržaj poruke. U gornjem slučaju je tip poruke bio `PROMJENA STANJA RACUNA`, a sadržaj je bio novo stanje računa. Na poruke se pretplaćuje korištenjem metode `pub.subscribe`, na način da joj se kao argumenti proslijede metoda koja će se izvršiti i poruka na koju se želi pretplatiti. Na ovaj način je doista moguće nezavisno razvijati pojedine dijelove aplikacije, jedino je potrebno voditi računa o porukama koje se šalju.

POGLAVLJE 4

Korišteni alati

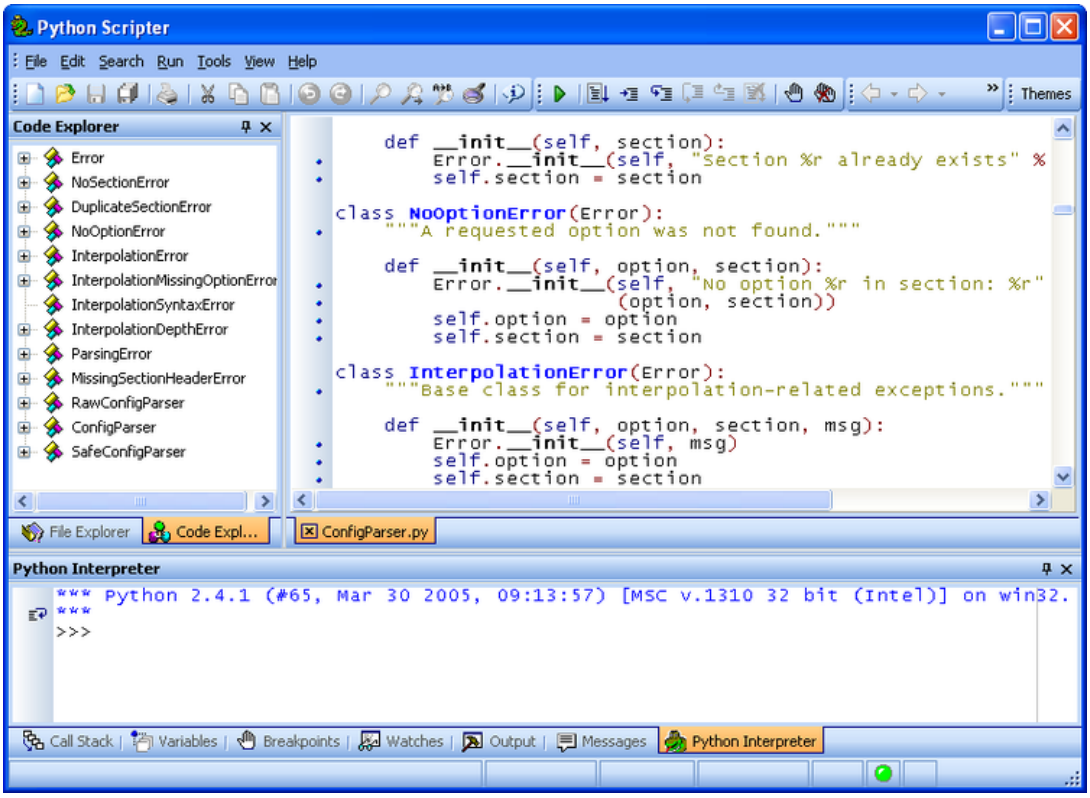
Sadržaj poglavlja

4.1 PyScripter

PyScripter je besplatna *Python* razvojna okolina (*Integrated Development Environment* - integrirano razvojno okruženje) otvorenog kôda. Zamišljen je kao alternativa komercijalnim alatima. Sa ostalim sličnim alatima stoji rame uz rame (npr. *Eclipse pyDevelop*). Podržava automatsko nadopunjavanje kôda, pregled kôda po klasama i funkcijama, isticanje sintakse (eng. *syntax hihglighting*) te je moguće otvoriti više *py* datoteka odjednom (u *tabovima*). Podržava obe Python račve (2.7 i 3+ *fork*).

4.2 Sphinx

Sphinx je alat koji je zamišljen da stvara vizualno privlačnu dokumentaciju inteligentno raspoređenog sadržaja. Isprva se koristio samo za *Python* dokumentaciju, ali su razvojem dodane nove mogućnosti tako da je ovo alat koji će kroz nekoliko godina biti izuzetno moćan.



Slika 4.1: Pyscripter IDE

Trenutno se kao markup jezik za pisanje dokumentacija koristi *RST* (*reStructuredText*) koji je izrazito pregledan jezik i pomoću Sphinx alata se jednostavno prevodi u cijeli niz drugih markup jezika.

Mogućnosti *Sphinx*-a su:

- **Izlazni format:** *HTML* (uključujući *Windows HTML help* datoteke), *LaTeX*, čisti tekst
- **Izrada križnih referenci (eng. *cross-reference*):** automatski se mogu napraviti tablice sa referencama, popisom funkcija, klasa, citata, ...
- **Hijerarhijska struktura:** lagano definiranje stabla dokumenta sa automatski izrađenim poveznicama na *parent* i *child* dijelove (primjerice <*h1*>,*<h2>*, ... tagove u *HTML*-u).
- **Rukovanje kôdom:** automatski se vrši isticanje sintakse
- **Dodaci:** mogućnost korištenja različitih dodataka

4.2.1 Instalacija

Sphinx je moguće instalirati na dva načina. Jednostavna metoda je korištenjem `easy_install` modula putem interneta gdje je putem konzole (*Command prompt*) dovoljno upisati (u poglavlju 2.1 su opisane moguće poteškoće):

```
cd putanja_do_Python_direktorija
cd Scripts
easy_install -U Sphinx
```

Nakon ovoga će `easy_install` sam skinuti *Sphinx* modul i instalirati ga. Ukoliko je putanja *Python* direktorija i njegovog poddirektorija *Scripts*, unešena u sistemsku *PATH* varijablu, prva dva koraka se mogu preskočiti.

Ukoliko internet veza nije dostupna, potrebno je sa službene stranice, otvoriti link "*Get Sphinx from the Python Package Index*" koji se nalazi sa desne strane. Nakon toga je potrebno odabrati za koju verziju Pythona je modul potreban i download počinje. Jednom kada je download gotov, modul se može instalirati na sljedeći način:

```
cd putanja_do_Python_direktorija
cd Scripts
easy-install -U putanja_do_download_direktorija\Sphinx-1.1.3-py2.7.egg
```

Kao i kod primjera prije, ukoliko je putanja *Python* direktorija i njegovog poddirektorija *Scripts*, unešena u sistemsku *PATH* varijablu, prva dva koraka se mogu preskočiti.

Kod *Windows* operativnog sustava, uz pretpostavku da je putanja Pythona `C:\Python27` i da se skinuta datoteka `Sphinx-1.1.3-py2.7.egg` nalazi u direktoriju `D:\Downloads`, instalacija bi se odvijala na sljedeći način:

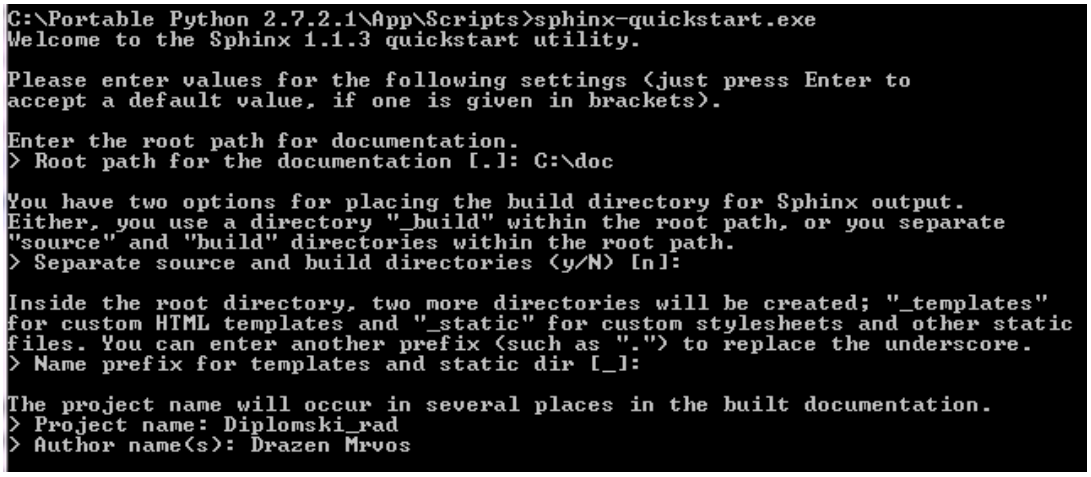
```
<otvori konzolu>
cd c:\Python27
cd Scripts
easy_install -U D:\Downloads\Sphinx-1.1.3-py2.7.egg
```

4.2.2 Izrada novog projekta

Nakon instalacije, u direktoriju *Scripts* je potrebno pokrenuti izvršnu datoteku `sphinx-quickstart.exe`. Tada će započeti interaktivna izrada novog projekta koje se svodi na odgovaranje na upite. Na kraju upita će u oblim zagradama pisati ponuđeni odgovori, a u uglatoj zagradi zadani odgovor. Na svaki upit se odgovara utipkavanjem odgovora (primjerice `y` ili `n`) i pritiskom tipke `Enter` nakon toga. Ukoliko se ne unese odgovor, a pritisne se tipka `Enter` - kao odgovor će se uzeti ono što je navedenu u uglatoj zagradi.

Otprilike je prva polovica odgovora bitna, a ta pitanja su redom sljedeća:

- putanja do *root* direktorija - *Root* direktorij je onaj u kojem će se napraviti direktorij sa svim potrebnim datotekama za dokumentaciju. On će sadržavati i korisnikove i generi-



Slika 4.2: Interaktivna izrada novog projekta

rane datoteke. Ukoliko se ništa ne unese, smatrati će se da je trenutni direktorij *Scripts* ujedno i *root* direktorij.

- da li je potrebno razdvojiti *source* i *build* direktorij - *Source* je direktorij u kojim se nalaze izvorne datoteke, a *build* je direktorij u kojem se nalazi sve što je Sphinx generirao korištenjem izvornih datoteki (primjerice *HTML* ili *LaTeX*). Ukoliko se odabere *n*, u *root* direktoriju će biti izvorne datoteke (korisnikove), a u njegovom poddirektoriju *build* će biti sve što je generirao *Sphinx*. Ako bi se odabralo *y*, onda bi se još napravio i poddirektorij source u kojem bi trebale biti sve izvorne datoteke. Dalje u tekstu se podrazumijeva da je kao odgovor uzet *y*.
- prefiks naziva predložaka - Svi *Sphinx* predlošci, a i korisnikovi vlastiti predlošci moraju imati neki prefiks. Ukoliko se ne navede drugačije, prefiks će biti "_"
- verzija i izdanje - potrebno je unjeti koja je verzija i koje izdanje projekta (npr. verzija 1, izdanje 1)
- ekstenzija izvornih datoteki - u pravilu `.rst` ili `.txt`
- naziv glavne datoteke - naziv datoteke koja je najviša po hijerarhiji, najbolje ostaviti zadanu vrijednost (*index*)

Nakon ovih odgovora je na ostale upite dovoljno odgovarati samo pritiskom na tipku *Enter*. Ukoliko će se u radu koristiti matematiče formule, potrebno je ipak obratiti pozornost na zadnjih nekoliko pitanja.

4.2.3 Opće postavke

U datoteci `conf.py`, koja se nalazi u direktoriju *source* (ili direktoriju *root* ukoliko se nije željelo razdvojiti *build* i *source* direktorij), se nalaze gotovo sve postavke koje su se odredilo

kroz interaktivnu izradu novog projekta. Datoteku je dovoljno otvoriti bilo kojim programom za uređivanje teksta.

Potrebno je pronaći liniju:

```
#language = None
```

I umjesto nje upisati sljedeće:

```
language = "hr"
```

Ovime ćemo *Sphinx*-u dati do znanja da sve što generira - generira na hrvatskom jeziku. Ovo se u pravilu odnosi na generirani HTML dokument.

4.2.4 Podešavanje LaTeX izvoza

Obzirom da je za diplomski rad potreban font *Times New Roman* veličine 12, potrebno je to nekako definirati. U datoteci `conf.py` je potrebno pronaći liniju:

```
latex_elements = {
        #neki key:value parovi
}
```

i taj cijeli blok zamijeniti sljedećim:

```
latex_elements = {
  'papersize': 'a4paper',
  'pointsize': '12pt',
  'fontpkg': '\\usepackage{times}',
  'classoptions': ',oneside',
  'babel' : '\\usepackage[croatian]{babel}',
  'title': u'Naslov rada',
  'author': u'Ime autora'
}
```

Sa tim blokom smo definirali sljedeće: dokument je prilagođen veličini A4 papira, veličina fonta je 12, a tip fonta je *Times New Roman*. Dokument je prilagođen jednostranom ispisu.

4.2.5 Generiranje HTML i TeX dokumenata

U *root* direktoriju se nalaz izvršna datoteka (točnije skripta) `make.bat`. Nju je potrebno pokrenuti sa jednim argumentom, a to je najlakše napraviti iz konzole. Nakon što se otvori konzola, potrebno je otići do *root* direktorija i zatim pokrenuti:

```
make html
make latexpdf
```

Nakon toga će se u *build* direktoriju generirati *HTML*, *LaTeX* dokument i PDF dokument iz *LaTeX* datoteki. Da se ne bi moralo svaki put otvarati konzolu i pokretati iz nje `make x`, može se u *root* direktorija napraviti datoteka `make_all.bat` i otvoriti je u nekom programu za uređivanje teksta (npr. *Notepad*). Nakon toga se u nju zapiše:

```
call make html
call make latex
```

Zatim se datoteka spremi, i svaki put kada je potrebno iznova generirati HTML/LaTeX dokument - dovoljno je nju pokrenuti, bilo putem konzole ili *Windows Explorera*.

4.3 Pandoc

Pandoc je odličan alat ukoliko je potrebno prevesti dokument iz jednog markup jezika u drugi. Trenutno je moguće raditi konverziju iz sljedećih formata:

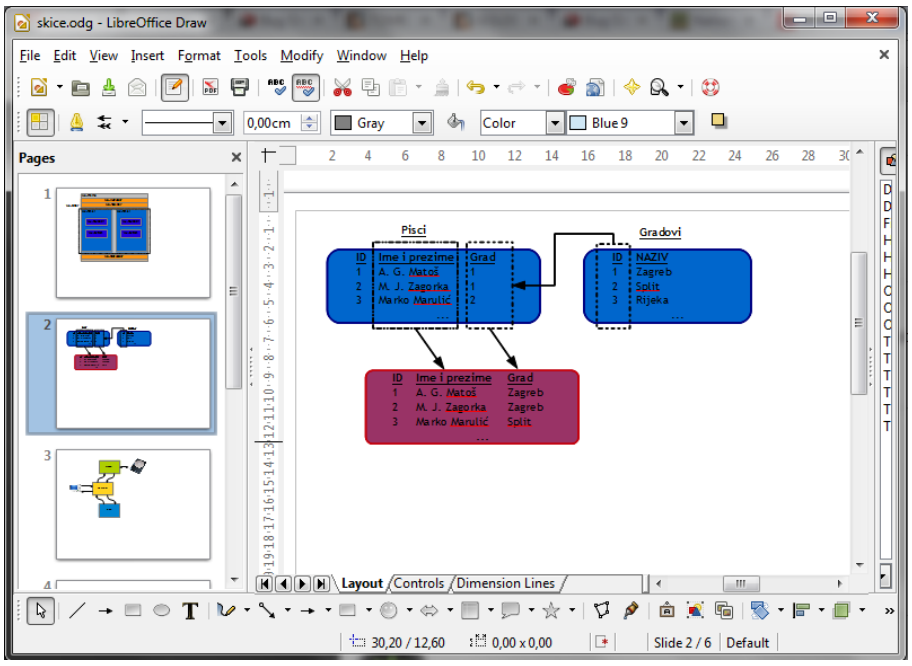
- **HTML formati:** XHTML, HTML5
- **Različiti formati Word procesora:** Microsoft Word docx, Open/LibreOffice ODT, OpenDocument XML
- **E-knjige:** EPUB
- **Formati za dokumentaciju:** DocBook, GNU TexInfo, Groff man
- **Tex formati:** LaTeX, ConTeXt, LaTeX Beamer
- **PDF** pomoću LaTeX-a
- **Ostali markup formati:** Markdown, reStructuredText (RST), AsciiDoc, MediaWiki, Emacs, Textile

Pandoc raspoznaje veliki broj dodataka pojedinih formata, uključujući meta podatke (naslov, autor, datum), fusnote, tablice, ... Ukoliko se želi, algoritmi raspoznavanja se mogu isključiti čime se dobiva čista konverzija.

4.4 LibreOffice Draw

Sve skice u radu su napravljene korištenjem *Draw* programa unutar *LibreOffice* paketa. *Draw* služi za izradu vektorske grafike, jednostavan je za korištenje, te je besplatan i otvorenog kôda

(kao i cijeli *LibreOffice* paket).



Slika 4.3: LibreOffice Draw sučelje

reStructuredText (reST)

Sadržaj poglavlja

reStructuredText je zamišljen kao jednostavan markup jezik zbog čega je njegovo korištenje produktivno i zahtijeva vrlo malo učenja.

5.1 Osnovni reST markup

5.1.1 Paragrafi

Osnovni blok u svakom reST dokumentu je paragraf. Paragrafi su blokovi teksta odvojeni jednom (ili više) praznom linijom. Uvlake su, kao i u Python-u, bitne tako da svi paragrafi moraju biti jednako poravnani sa lijeve strane.

5.1.2 Inline Markup

Osnovni inline markup je poprilično jednostavan:

- jedna zvjezdica: `*primjer*` za naglašavanje (eng. *emphasis*, odnosno italic): *primjer*
- dvije zvjetdice: `**primjer**` za jako naglašavanje (*bold*): **primjer**

- otvoreni polunavodnici (eng. *backquotes*): ``` ``primjer`` ``` za pisanje kôda (*verbatim* okolina): `primjer`

Naravno, postoje neka ograničenja vezana uz ovaj markup:

- ne mogu se ugnjezditi
- sadržaj ne smije počinjati ili završavati sa razmakom: `* primjer` je krivo napisan
- mora se odvojiti od ostatka teksta pomoću znaka koji se ne prikazuje (primjerice prekid linije i sl.). Ovo se može zaobići na način da se stavi kosa crta u lijevo prije razmaka: `Ovoje\ *jedna*\ riječ`, što daje Ovoje*jedna*riječ

Osim navednog načina, moguće je navesti i markup za cijeli sadržaj. Ta se sintaksa ujedno koristi i za križne reference (eng. *cross-reference*, pojašnjeno kasnije). Opća sintaksa je sljedeća: `:uloga:`sadržaj``. Popis uloga je sljedeći:

- *emphasis* - alternativni zapis za `*emphasis*`
- *strong* - alternativni zapis za `**strong**`
- *literal* - alternativni zapis za ``` ``literal`` ```
- *subscript* - tekst koji ide u indeks
- *superscript* - tekst koji ide kao eksponent

5.1.3 Liste

Sintaksa za liste je intuitivna: dovoljno je staviti zvjezdicu na početak paragrafa i paziti na uvlake. Isto vrijedi i za pobrojane liste, koja ujedno mogu biti i automatski numerirane koštenjem znaka # (eng. *hash*, ljestvice):

```
* Obična lista.
* Ova lista ima dva predmet, drugi
  zauzima dva retka

1. Ovo je pobrojana lista.
2. Ima tri predmet.
4. Treći predmet je preskočio broj

# Ovo je također pobrojana lista,
# ali se ovdje ne treba brinuti oko numeriranja
```

Gore navedeni kôd daje sljedeće:

- Obična lista.

- Ova lista ima dva predmet, drugi zauzima dva retka

1. Ovo je pobrojana lista.
2. Ima tri predmeta.
3. Zadnji predmet je automatski numeriran

Liste je moguće ugnjezditi, podliste se definiraju dodatnom uvlakom u odnosu na osnovnu listu. Sljedeći kôd:

```
* sada slijedi nadasve
* zanimljiva lista

  * sa podlistom koja kreće ovdje
  * a završava ovdje

* i nakon podliste se osnovna lista nastavlja ...
```

će kao rezultat dati:

- sada slijedi nadasve
- zanimljiva lista
  - sa podlistom koja kreće ovdje
  - a završava ovdje
- i nakon podliste se osnovna lista nastavlja ...

Još jedan tip listi su *definicijske liste*. Svaki predmet definicijske liste se sastoji od pojma, klasifikatora (neobavezno) i definicije (opisa). Pojam je jednolinijska riječ ili izraz. Klasifikatori se definiraju iza pojma, u istoj liniji i to nakon " : " (razmak, navodnik, razmak). Definicija je blok uvučen u odnosu na pojam i može sadržavati više paragrafa ili drugih elemenata. Između definicijskog bloka i linije sa pojmom ne smije biti praznih redova. Prazne linije su nužne prije prvog i nakon zadnjeg predmeta liste, između se stavljaju po volji:

```
pojam 1
        Definicija 1.

pojam 2
        Definicija 2, paragraf 1.

        Definicija 2, paragraf 2.

pojam 3 : klasifikator
        Definicija 3.
```

```
pojam 4 : klasifikator prvi : klasifikator drugi
        Definicija 4.
```

Gore navedeni kod će dati:

pojam 1 Definicija 1.

pojam 2 Definicija 2, paragraf 1.

Definicija 2, paragraf 2.

pojam 3 [klasifikator] Definicija 3.

pojam 4 [klasifikator prvi][klasifikator drugi] Definicija 4.

Definicijske liste imaju nekoliko primjena:

- kao rječnik ili pojmovnik. Sama riječ je pojam, klasifikator tip (imenica, glagol, ...) i definicija dolazi iza njih.
- za opisivanje varijabli u programu. Pojam je u tom slučaju ime varijable, klasifikator tip varijable (int, string, ...), a definicija opisuje na koji se način koristi.

Kako bi se dobio paragraf za citat, dovoljno je samo dio koji spada pod citat, uvući više nego okolne paragrafe.

Ukoliko je potrebno sačuvati prelom redova (eng. *line-break*), dovoljno je ispred svakog reda staviti okomitu crtu. Primjer:

```
| Prijelom ovih redova
| će u dokumentu biti
| potpuno jednak kao i u ovome kodu
```

5.1.4 Prikazivanje izvornog kôda

Ukoliko je potrebno ubaciti dio preformatiranog teksta kojeg se nipošto ne smije mijenjati niti shvaćati njegove dijelove kao tagove, dovoljno je paragraf prije završiti sa `::` (dvostruka dvotočka). Blok sa kôdom mora biti uvučen u odnosu na paragraf prije i nakon njega. Sljedeći kod:

```
Primjer::

   Različiti dijelovi **kôda** \\ se doslovce prikazuju
         bez da se brine o tome hoće li :ref: ili

   * ovo
```

```
   * ovo
   * ovo

   Završiti kao referenca ili lista
```

Rezultira sljedećim:

Primjer:

```
Različiti dijelovi **kôda** \\ se doslovce prikazuju
               bez da se brine o tome hoće li :ref: ili

* ovo
* ovo
* ovo

Završiti kao referenca ili lista
```

Oznakom `::` se pametno rukuje:

- ukoliko se pojavi kao paragraf za sebe, taj se paragraf u potpunosti izostavlja iz dokumenta.
- ukoliko se prije njega nalazi razmak, oznaka se uklanja.
- ukoliko se prije njega nalazi bilo koji znak (osim razmaka), oznaka se zamijenjuje jednostrukom dvotočkom. Kao što se vidi u gornjem primjeru `Primjer::` postaje `Primjer:`.

5.1.5 Tablice

Postoje dvije vrste tablica: jednostavne tablice (eng *Simple tables*) i tablica sa mrežom (eng. *Grid tables*). Jednostavne je lakše za pisati ali su ograničene mogućnostima. Moraju obavezno imati više od jednoga reda, i prvi stupac ne smije imati više od jednog retka. Primjerice, kôd:

```
=====  =====  =======
    Ulazi       Izlaz
-----------  -------
=====  =====  =======
0      0      0
1      0      0
0      1      0
1      1      1
=====  =====  =======
```

daje:

Ulaz		Izlaz
A	B	A AND B
0	0	0
1	0	0
0	1	0
1	1	1

Ovaj tip tablica je opisan horizontalnim granicama koje čine znakovi "=" i "-". Znak jednako ("=") se koristi za definiranje vrha i dna tablice, te za odvajanje redaka zaglavlja od ostatka tablice. Minus ("-") se koristi da se naznače ćelije koje se protežu kroz nekoliko stupaca (eng. *column span*) - ćelija "Ulaz" iz gornjeg primjera. Također, "-" se po želji može koristiti i kako bi se eksplicitno i vizualno razdvojili retci.

Kao što se u gornjem primjeru i vidi, jednostavna tablica počinje sa gornjim rubom koji se sastoji od "=" znakova, odvojenih razmakom (dva ili više se preporučuje) na rubovima stupaca. Bez obzira na kasnije protezanje ćelija kroz nekoliko stupaca, gornji rub *mora* u potpunosti opisati sve stupce tablice. U tablici moraju postojati barem dva stupca. Nakon gornjeg ruba može ići redak zaglavlja, ali nakon tog retka obavezno idu znakovi "=", sukladno sa rubovima stupaca. Donji rub tablice se također sastoji od znakova "=", također sa razmacima na rubovima stupaca.

Prazni redovi su dozvoljeni unutar jednostavnih tablica. Njihova interpretacija ovisi o kontekstu. Prazni redovi između redaka tablice se ignoriraju. Ukoliko se redak tablice sastoji od više linija, umetanje praznih redova može odvojiti paragraf. Krajnje desni stupac je neograničen - tekst se može nastaviti i nakon ruba tablice. Unatoč tome, preporuča se da se rubovi tablice postave tako da obuhvaća cijeli tekst.

Ako se koriste tablice sa mrežom, potrebno je "nacrtati" cijelu tablicu (cijelu mrežu). Tablice sa mrežom pružaju cjelovit prikaz tablice poput "*ASCII slika*". U ćelije se mogu stavljati drugi elementi (npr. liste ili slike), a ćelije se mogu protezazi kroz više redaka i/ili stupaca. Tablice sa mrežom mogu biti poprilično zamorne za napraviti pa se preporučuje korištenje jednostavnih tablica ukoliko je to moguće. Primjer:

```
+------------------------+------------+-------------+
| Zaglavlje              | Zaglavlje 2 | Zaglavlje 3 |
| (redak u zaglavlju   ) |             |             |
+========================+============+=============+
| redak 1, stupac 1      | stupac 2    | stupac 3    |
+------------------------+------------+-------------+
| redak 2                | ćelija kroz 2 stupca      |
+------------------------+------------+-------------+
```

Dobiva se:

Zaglavlje (redak u zaglavlju)	Zaglavlje 2	Zaglavlje 3
redak 1, stupac 1	stupac 2	stupac 3
redak 2	ćelija kroz 2 stupca	

5.1.6 Hiperlinkovi

Vanjski linkovi se mogu unjeti u tekstu koristeći `'Zrno <http://zrno.fsb.hr/>'_`, što daje: Zrno - poveznicu na stranicu. Ukoliko je tekst koji se treba prikazati ujedno i sama web adresa, nije potrebna nikakva posebna oznaka - dovoljno je samo upisati adresu: `http://zrno.fsb.hr` daje http://zrno.fsb.hr.

Ukoliko se želi odvojiti definiranje hiperlinka od teksta, moguće je napraviti i sljedeće: bilo gdje u dokumentu se definira adresa na sljedeći način: `.. _a link: http://primjer.com/`. Navedeno je referenca, može se definirati bilo gdje u dokumentu ali se obično definiraju na kraju poglavlja ili dokumenta. Sljedeći primjer pokazuje čemu to točno služi:

```
Paragraf koji sadrži 'link'_.

.. _link: http://zrno.fsb.hr/
```

Gornji primjer daje sljedeće:

> Paragraf koji sadrži link.

Kao što se vidi, link se definira kao referenca koja se može bilo gdje u dokumentu pozvati (recimo, varijabla koja sadrži adresu).

5.1.7 Cjeline

Zaglavlja cjelina se definiraju podvlačenjem (moguće i nadvlačenjem) naslova cjeline sa punktacijskim znakom, barem u dužini naslova:

```
##############
Ovo je cjelina
##############

Struktura i dubina zaglavlja je definirana pojavom novih zaglavlja:

######
Naslov
######
```

```
Znak "#" se prvi pojavio u definiciji zaglavlja pa će se njime označavati prva r

Podnaslov
=========

Znakom "=" će biti definirana druga razina cjelina

...
```

Iako je korisniku na volju da odabere redoslijed znakova koji označuju pojedine cjeline, postoji dogovor koji je u pravilu dobro poštivati:

- # sa nadvlakom za dijelove
- * sa nadvlakom za poglavlja
- = za naslove
- − za podnaslove
- ^ za podpodnaslove
- " za odlomke

5.1.8 Dodavanje slika

Dodavanje slika je jednostavno, dovoljno je koristiti sljedeći kôd:

```
.. image:: naziv_slike.x
        opcije
```

Slika ne mora biti u istom direktoriju ali je tada potrebno navesti putanju do nje, bilo relativnu (npr. `slike/fsb-png`) ili apsolutnu (npr. `c:\a\b\c\proba.png`).

Sphinx će prilikom izrade dokumentacije sve slike kopirati u zaseban direktorij (primjerice `_static` direktorij izlaznog *HTML* direktorija).

Sphinx** dodaje vlastitu mogućnost za dodavanje slika:

```
.. image:: gnu.*
```

Ovom naredbom *Sphinx* pretražuje sve slike koje odgovaraju zadanom nazivu. Prilikom svakog izvoza zatim odabire najprigladniji tip datoteke ukoliko ih ima više. Primjerice, ukoliko se sa uzorkom `gnu.*` nađu dvije datoteke: `gnu.pdf` i `gnu.png`, pdf će se koristiti za *LaTeX* izvoz, a png za *HTML* izvoz.

5.1.9 Fusnote

Za definiranje fusnote je potrebno koristiti `[#naziv]_` kako bi označili točnu lokaciju fusnote, a zatim je na kraju dokumenta u `Footnotes` zaglavlju potrebno navesti sadržaj fusnote:

```
Pišem tekst [#f1]_ i označujem fusnote [#f2]_

.. rubric:: Footnotes

.. [#f1] Tekst prve fusnote
.. [#f2] Tekst druge fusnote
```

Fusnote je moguće numerirati eksplicitno (`[1]_`) ili koristiti auto-numeriranje fusnota bez naziva (`[#]_`).

5.1.10 Citati

Reference citata su globalne, tj. moguće ih je referencirati iz bilo kojeg povezanog dokumenta. Definiraju se jednostavno:

```
Ovo je citat [Ref]_ za koji želim definirati izvor.

.. [Ref] Neka knjiga, članak, URL, ...
```

Citati se koriste na sličan način kao i fusnote, razlika je jedino u tome što moraju imati naziv.

5.1.11 Komentari

Svaki blok koji nema navedeni markup konstruktor (npr. `[Ref]`, ili `link_`) se smatra komentarom:

```
.. ovo je komentar
```

Ukoliko je potrebno staviti komentar koji se proteže kroz nekoliko redova teksta, dovoljno ga je uvući, odnosno napraviti zaseban blok za njega:

```
..
        Ovo je malo
        duzi komentar.

        I dalje komentiram
```

5.2 Sphinx markup unutar RST dokumenta

5.2.1 Sadržaj (TOC tree)

Obzirom da u *reST* dokumentima ne postoji mogućnost povezivanja nekoliko različitih dokumenata, ili podijele dokumenta u nekoliko datoteka, *Sphinx* koristi vlastite naredbe kako bi se postavio odnos između pojedinih datoteka u dokumentaciji. Time se ujedno izrađuje i stablo sadržaja.

Izrada stabla sadržaja

Naredba `toctree` ubacuje stablo sadržaja na trenutnu lokaciju u dokumentu, korištenjem zasebnih `toc` naredbi (uključujući i pod-stabla). Relativni nazivi dokumenata (ne počinju sa kosom crtom u desno) su relativni trenutnom direktoriju dokumenta. Moguće je unijeti i apsolutnu putanju do pojedinih dokumenata na način da se jednostavno upiše puna putanja do dokumenta. Opcija `maxdepth` se koristi kako bi odredili do koje dubine zaglavlja će biti prikazano u stablu:

```
.. toctree::
   :maxdepth: 2

   uvod
   razrada
   diskusija
   zakljucak
   (ostali dokumenti)
```

Gore navedeni primjer služi za dvije stvari:

- Ubacuje se sadržaj iz svih dokumenata sa najvećom dubinom 2, što znači da će se prikazati glavno zaglavlje i jedno ugnježđeno. `toctree` naredbe u tim dokumentima se također uzimaju u obzir.
- *Sphinx*-u na ovaj način definiramo redoslijed dokumenata i dajemo do znanja da se oni nalaze unutar prikazanog dokumenta.

Naslovi u sadržaju

U gornjem primjeru će stablo sadržaja uvesti navedene datoteke (uvod, razrada, ...) i prikazati naslove svih zaglavlja do navedene dubine. No, ukoliko se ne želi koristiti te naslove nego

neke vlastite, dovoljno je prije naziva dokumenta staviti željeni naslov i zatim naziv dokumenta navesti u izlomljenim (< >) zagradama:

```
.. toctree::
   :maxdepth: 2

   Uvod u XY <uvod>
   Praktični dio <razrada>
   Rezultati <diskusija>
   zakljucak
   (ostali dokumenti)
```

Numeriranje sadržaja

Gore navedena stabla sadržaja neće biti numerirana. Ukoliko se želi numerirati stabla sadržaja, potrebno je dodati `numbered` opcija:

```
.. toctree::
   :maxdepth: 2
   :numbered:

   dokumenti
```

Opcija `numbered` bi se trebala koristiti samo na glavnom sadržaju.

Uvoz više datoteka

Kako bi se olakšao uvoz datoteka koje imaju zajednički prefiks, koristi se opcija `glob`:

```
.. toctree::
   :glob:

   uvod*
   razrada/*
   *
```

Gornji primjer će uvesti sve datoteke sa prefiksom *uvod*, zatim sve datoteke iz direktorija *razrada* i zatim sve preostale datoteke (osim glavne naravno). Ukoliko se baš želi navesti i glavna datoteka, može se na način da je se navede sa *self*:

```
.. toctree::

   self
```

Sakrivanje datoteki iz sadržaja

Obzirom da postoji mogućnost da korisnik ne želi staviti neke dokumente u stablo sadržaja, ali želi odrediti hijerarhiju dokumenta, koristi se opcija `hidden`. Donji primjer će datoteku uvod uvesti u sami dokument ali ju neće navesti u sadržaju:

```
.. toctree::
   :hidden:

   uvod
```

5.2.2 Markup paragrafa

Postoji nekoliko oznaka kojima se određeni dio teksta naglašava.

Napomena (*note*)

Ako se želi naglasiti određeni dio dokumentacije, moguće je određeni dio postaviti unutar PRAVOKUTNIKA kako bi se dio istaknuo.

Primjer:

```
.. note::

      Ovo je jedna vrlo važna napomena
```

Rezultat:

Napomena: Ovo je jedna vrlo važna napomena

Upozorenje

Slično napomeni, i ova oznaka ističe tekst ali sa oznakom **Upozorenje**. Primjer:

```
.. warning::

      Ovo je upozorenje
```

Rezultat:

Upozorenje: Ovo je upozorenje

Prikazivanje izvornog kôda

Označavanje izvornog kôda je podržano za čitav niz jezika (*Python*, *Ruby*, *C*, *ReST*, ...). Za razliku od *čistog reST* jezika, u *Sphinx* dokumentima je pretpostavka da nakon oznake `::` slijedi izvorni kod. Ako se drugačije ne navede, *Sphinx* pretpostavlja da se radi o *Python* kôdu i odgovarajuće će ga označiti. Ako se ne radi o *Python*-u nego o nekom drugom jeziku, to se može promijeniti na sljedeći način:

```
.. highlight:: c
```

Gornjim primjerom se postiglo da od te točke na dalje, *Sphinx* pretpostavlja da se iza svake `::` oznake, nalazi *C* kôd i tako će ga i označiti. Ako napisani kôd ne odgovara kôdu jezika koji je naveden u zadnjem `highlight` dijelu, kôd se neće označiti nego će ostati kao običan komentar.

Ukoliko je u dokumentu potrebno pisati primjere iz više programskih jezika, bilo bi nezgrapno svaki čas mijenjati `highlight` opciju pa iz tog razloga postoji `code-block` okolina u kojoj se direktno navodi iz kojeg programskog jezika je kôd:

```
.. code-block:: python

        print "Python kôd"
        print "Još python kôda"
```

Argumenti koji se navodi u `highlight` i `code-block` dijelu može biti:

- *none* - bez označavanja
- *python* - ukoliko se sa `highlight` blokom ne odredi drugačije, koristiti će se ovo označavanje
- *guess* - Sphinx će sam pokušati odrediti o kojem se programskom jeziku radi
- rest - za *reST* kôd
- c
- bilo koji drugi podržani ...

Ako se u dokumentu navode dugački kôdovi, biti će poprilično komplicirano referencirati se na njih - primjerice ukoliko se nešto zanimljivo nalazi u redu 100, nije podobno pustiti korisnika da broji do tog reda. Iz tog razloga se može uključiti numeracija redova u primjerima kôda.

Ukoliko se koristi automatsko označavanje kôda (oni koji počinju sa `::`), to se može postići na sljedeći način:

```
.. highlight:: python
   :linethreshold: 5
```

Gornji kôd će dati do znanja interpreteru da u svim primjerima koji imaju više od 5 linija kôda se postave brojevi linije.

Ako se koristi `code-block` okolina, dovoljno je unutar nje postaviti `lineons` opciju:

```
.. code-block:: python
   :lineons:

   (kôd)
```

Kako bi se moglo još jasnije referencirati na primjer kôda, postoji i opcija `emphasize-lines` unutar `code-block` okoline. Ta opcija ističe zadane dijelove kôda. Primjer:

```
.. code-block:: python
 :emphasize-lines: 3,5

 def some_function():
    dobar_primjer = True
    print 'Naglašena linija.'
    print 'Nenaglašena linija.'
    print 'Još jedna naglašena!.'
```

Prethodni primjer će kao rezultat dati:

```
def some_function():
  dobar_primjer = True
  print 'Naglašena linija.'
  print 'Nenaglašena linija.'
  print 'Još jedna naglašena!.'
```

Prilikom navođenja većih dijelova kôda, sami *reST* dokument bi postao previše nezgrapan. Iz tog razloga je pomoću *Sphinx*-a moguće u *reST* dokument direktno ubaciti vanjsku datoteku sa kôdom:

```
.. literalinclude:: primjer.py
```

Naziv datoteke je relativan putanji datoteke u kojoj se uvozi. No, moguće je navesti i apsolutnu putanju (počinje sa /) koja je relativna *source* direktoriju projekta. Ova okolina podržava

`lineons` opciju za prikaz broja linije kôda, `emphasize-lines` za naglašavanje linija i `language` opciju za definiranje trenutnog jezika:

```
.. literalinclude:: primjer.c
   :language: c
   :emphasize-lines: 12,15-18
   :linenos:
```

Ukoliko je datoteka sa izvornim kôdom drugačijeg kodiranja (eng. *encoding*) od *reST* dokumenta, moguće je definirati kodiranje na sljedeći način:

```
.. literalinclude:: primjer.py
   :encoding: latin-1
```

Vrlo korisna mogućnost je uključivanje pojedinih dijelova kôda iz *Python* datoteki. Primjerice, moguće je uvesti samo neku klasu, funkciju ili metodu:

```
.. literalinclude:: primjer.py
   :pyobject: Brisi.sve
```

Gore navedeni kôd će iz *Python* datoteke *primjer* uvesti metodu *sve* iz klase *Brisi*. Ukoliko se ne radi o *Python* datoteki, ili se iz *Python* datoteke žele samo određene linije kôda uvesti, potrebno je koristiti opciju `lines`:

```
.. literalinclude:: primjer.py
   :lines: 2,5,7-10,15-
```

Gornji primjer će prikazati samo linije 2, 5, 7 do 10 i sve od 15 do kraja dokumenta *primjer.py*.

5.3 Pisanje reST dokumenta

Za pisanje *reST* dokumenta nije potreban nikakav poseban program, dovoljno je koristiti bilo kakav program za obradu teksta kao što je *Notepad* koji dolazi uz Windowse ili primjerice *gEdit* ukoliko se radi o Ubuntu okruženju.

POGLAVLJE 6

Aplikacija za označavanje i obradu slika

Sadržaj poglavlja

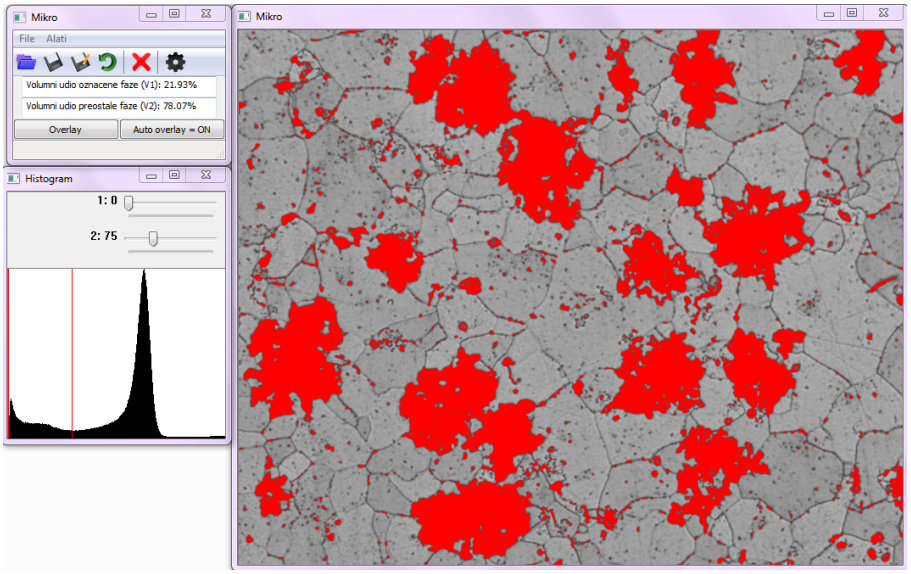
Cilj izrade aplikacije za označavanje i obradu slika je korisniku pružiti jednostavan način osnovne obrade uvezenih slika, te njihovo označavanje. Ovakva aplikacija bi bila vrlo korisna osobama koje se bave digitalizacijom fotografija, knjiga i sl., na način da korisnik može pregledavati slike i stavljati markere na mjesta gdje je skenirana knjige primjerice oštećena. Ukoliko je kvaliteta loša, postavlja se neki drugi marker i sl. Na ovaj način se zamoran posao, pregledavanja i zapisivanja problema na komad papira (ili u *Notepad*) olakšava i ubrzava vizualnim prikazom problema/napomena.

Korisnik može dodati jednu ili više slika i spremiti ih kao jedan slijed (niz) slika. Ista slika može biti u više različitih slijedova. Sve oznake koje se postave na sliku se ne smiju spremati na izvornu sliku, ali se u svakom slučaju moraju spremiti na neki drugi način.

Uz sve to, oznake se mogu rasporediti po slojevima koje korisnik sam definira za svaki slijed. Trenutni prikaz slike se mora mijenjati u ovisnosti o trenutno odabranim slojevima.

6.1 Povijest razvoja aplikacije

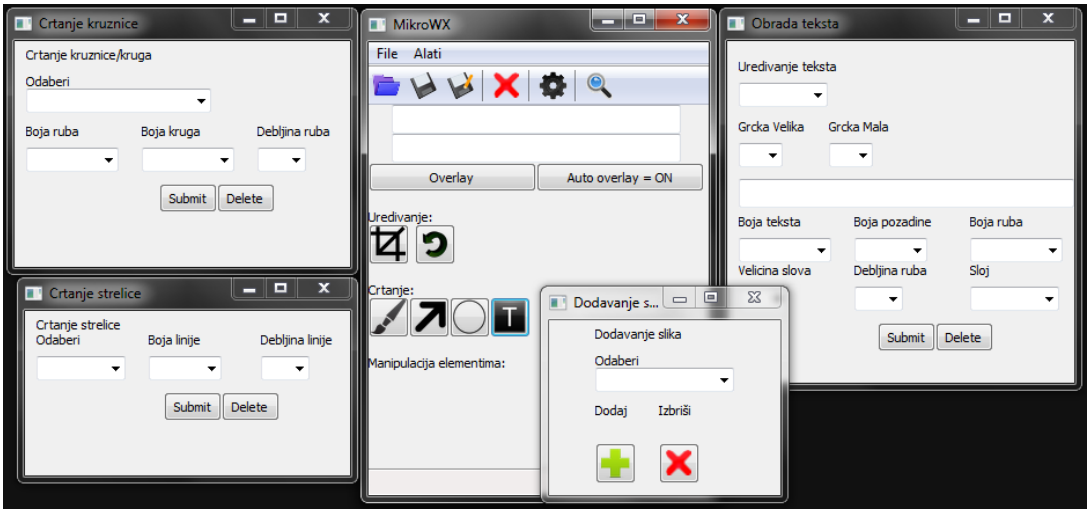
Prva verzija aplikacije je razvijena za potrebe završnog rada. Zahtjev prve verzije je bio da postoji mogućnost osnovne obrade slike - rezanje, mijenjanje oštrine, kontrasta i svjetline. Glavna zadaća te aplikacije je bilo određivanje volumnog udjela fazi pojedinih slitina, pomoću definiranja razine sivila (*grayscale*) u odnosu na cijelu sliku. Cijela aplikacija je napravljena korištenjem OpenCV i wxPython paketa.



Slika 6.1: Korisničko sučelje aplikacije razvijene za potrebe završnoga rada

Nakon završnog rada se aplikacija nastavila razvijati za potrebe projekta na diplomskom studiju. Zahtjev ove aplikacije je bio da se omogući postavljanje različitih oznaka na sliku, crtanje osnovnih likova, ... Glavna razlika u odnosu na verziju završnoga rada je korištenje *SQLite* baze kako se promjene napravljene nad slikom ne bi spremale na samu sliku, nego u bazu podataka. Naravno, kopija slike se sa svim promjenama mogla po potrebi spremiti na zasebnu lokaciju. Uz to, obzirom na nedostatke *OpenCV*-a, dodana je mogućnost povećanja slika i micanja po njoj (eng. *Zoom and Pan*). Uz *OpenCV* i *wxPython* paket, korišten je još paket *Python Imaging Library - PIL*.

Kao što se može vidjeti na slikama prvih verzija aplikacije, postojao je veliki broj prozora, odnosno jedan za svaku pojedinu funkciju. Ti svi prozori su bili napravljeni putem *wxPython* paketa. Prozor za prikaz same slike je dio *OpenCV* paketa, što znači da su različiti događaji koji se mogu pozvati, kao i dostupni elementi grafičkog sučelja različiti u odnosu na *wxPython*. Obzirom na to, nije postojala mogućnost ujedinjavanja cijelog sučelja kako bi se dobilo sučelje kakvo imaju sve modernije aplikacije (*ACDSee*, *XNView*, *Adobe Photoshop*, ...). Iz tog razloga se odustalo od korištenja OpenCV paketa. Grafički prikaz je u potpunosti preuzeo *wxPython*, a funkcionalni dio je u potpunosti preuzeo već spomenuti *PIL*. Na ovaj način se

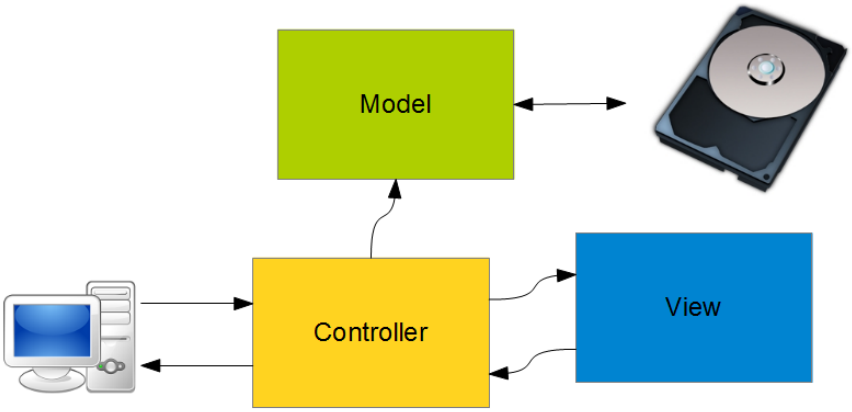


Slika 6.2: Korisničko sučelje aplikacije razvijene za potrebe projekta na diplomskom studiju

postigla funkcionalnost kao i u prethodnim verzijama ali uz puno kompaktinije i oku ugodnije grafičko korisničko sučelje. Naglasak aplikacije je stavljen na označavanje slika, što će kasnije biti puno detaljnije opisano.

6.2 Primijenjena MVC arhitektura

Kod klasične *MVC* arhitekture *Model* šalje obavijesti prilikom promjene baze. Kod *WEB* aplikacije, gdje se može spojiti više korisnika u isto vrijeme, to i ima smisla. Obzirom da je napravljena aplikacija namijenjena za rad na osobnom računalu i lokalnom bazom podataka, ovo nije potrebno. To znači da jedino *Controller* započinje bilo kakvu vrstu komunikacije sa bazom, a baza ne šalje nikakve notifikacije o promjeni.

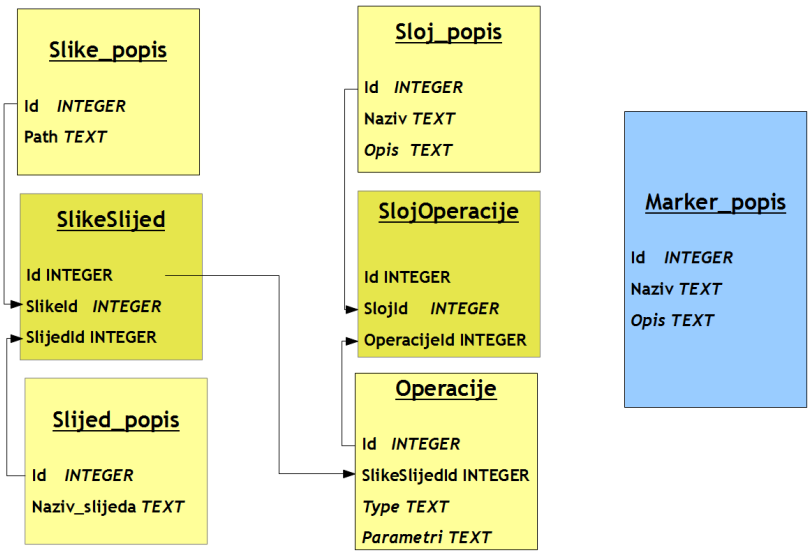


Slika 6.3: Primijenjena MVC arhitektura

6.3 Struktura baze podataka

Baza je osmišljena tako da se podaci u njoj ne prikazuju dva puta, odnosno tamo gdje je moguće se koriste reference korištenjem `FOREIGN KEY` atributa. U bazi se nalaze sljedeće tablice:

- *Slike_popis* - sadrži popis svih slika koje su dodane na način da se upisuje njihova putanja
- *Slijed_popis* - svaki naziv niza slika se zapisuje u ovu tablicu
- *SlikeSlijed* - obzirom da svaka slika može biti u više slijedova, dolazi se do odnosa *many-to-many* koji se rješava kreiranjem dodatne tablice (eng. *junction table*) koja povezuje identifikacijske oznake slika i slijedova.
- *Sloj_popis* - sadrži popis slojeva koji vrijede za svaki slijed
- *Operacije* - najvažnija tablica koja sadrži informacije o svakom elementu koji se dodao na sliku.
- *SlojOperacije* - dodatna tablica koja povezuje popis slojeva i popis operacija za određenu sliku u određenom slijedu
- *Marker_popis* - sadrži popis svih markera koje je moguće dodati na slike



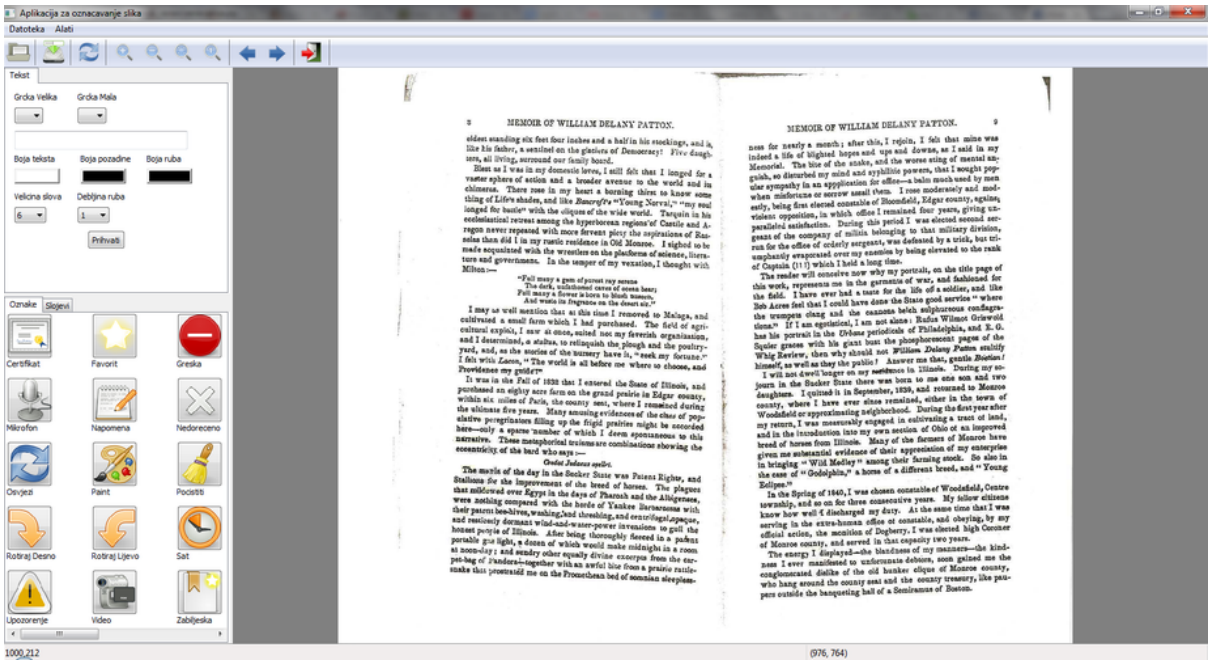
Slika 6.4: Struktura korištene baze podataka

6.4 Opis rada aplikacije i njenih pojedinih dijelova

Ukoliko se pogledaju aplikacije razvijene kroz završni rad i projekt, vidljivo je da sučelje nije kompaktno. U tim aplikacijama drugačiji izgled i nije bio moguć, obzirom da je za prikaz slike

korišten sami `OpenCV`. Prilikom izrade diplomskog rada je odlučeno da će aplikacija imati oku ugodniji i profesionalniji izgled. Iz tog razloga je, nakon nekoliko preinaka, osmišljeno sučelje koje je modularno - novi alati se jednostavno dodaju, stari se jednostavno mijenjaju. Aplikacija je iz tog razloga raspodijeljena na nekoliko *python* datoteki:

- `BaseClass.py` - ovo je `model` klasa, služi za komunikaciju sa bazom podataka
- `Draw.py` - sadrži sve funkcije za crtanje - primjerice teksta na slike
- `Frames.py` - sadrži veliki dio elemenata korisničkog sučelja - uglavnom dijaloge
- `ImageClass.py` - klasa koja brine oko uvećavanja i umanjivanja slika. U starijim verzijama aplikacije se brinula i oko pomicanja slike (eng. *Pan*), što je u ovoj verziji riješeno prozorom sa klizačima
- `Main.py` - sadrži temeljnu `View` i `Controller` klasu



Slika 6.5: Korisničko sučelje

Korisničko sučelje se sastoji od 4 osnovna dijela. Glavni izbornik (sa izbornicima *Datoteka* i *Alati*), alatna traka, dio za dodavanje elemenata (cijeli lijevi dio ispod alatne trake) i dio za prikaz slike.

6.4.1 Glavni izbornik

Glavni izbornik se sastoji od dva izbornika: izbornik *Datoteka* koji sadrži funkcije vezane uz upravljanje datotekama, i izbornik *Alati* koji sadrži alate za upravljanje pojedinim funkcijama.

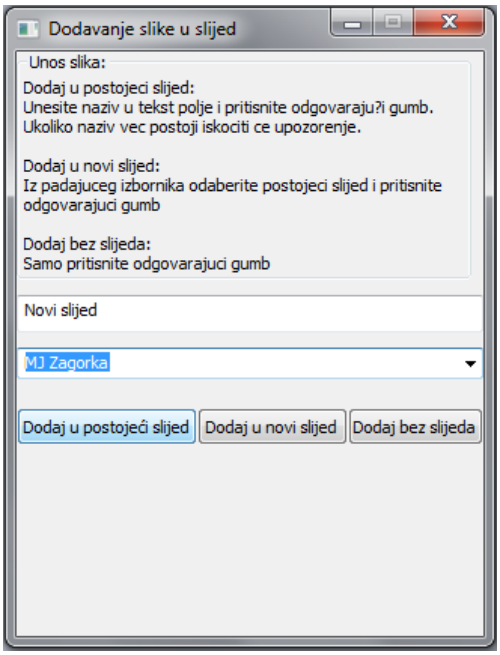
Izbornik *Datoteka* sadrži predmete:

- *Dodaj sliku u bazu* - pojavljuje se dijalog za odabir jedne datoteke
- *Dodaj direktorij u bazu* - pojavljuje se dijalog za odabir direktorija, a sve slike iz njega se unose u bazu
- *Dodaj direktorij sa poddirektorijima u bazu* - kao i prethodni element, razlika je što se slike dodaju i iz svih poddirektorija
- *Otvori postojeći slijed* - otvara neki od postojećih slijedova u bazi
- *Spremi kopiju* - sprema se kopija slike sa svim elementima na sebi
- *Izađi* - Izlazak iz programa

Izbornik alati sadrži:

- Uređivanje oznaka - pojavljuje se jednostavan prozor iz kojeg je moguće dodati novu oznaku u bazu
- Uređivanje slojeva - pojavljuje se jednostavan prozor u kojem je moguće odabrati sloj za brisanje ili dodati novi sloj

Ukoliko korisnik odluči dodati bilo koju sliku u bazu (pojedinačnu ili cijeli direktorij), nakon što ju odabere pojaviti će se prozor iz kojeg je potrebno odrediti pripadnost slike nekom slijedu. Slika/niz slika se mogu dodati u postojeći slijed, novi slijed ili bez slijeda (korisno primjerice ukoliko se želi samo pogledati slika).



Slika 6.6: Dodavanje slike ili niza slika u slijed

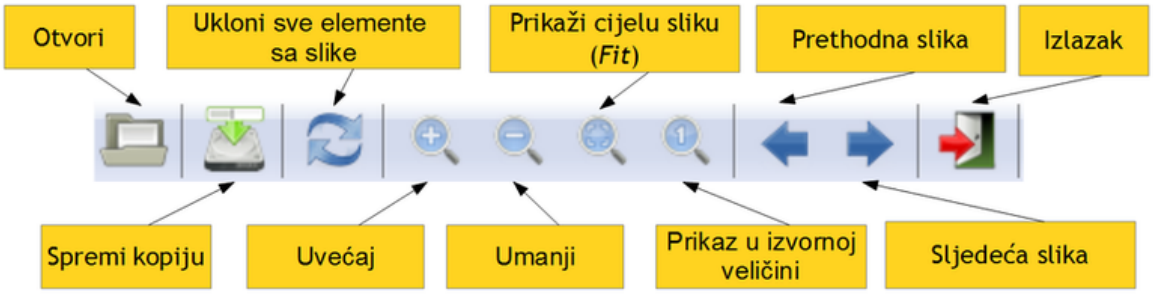
Nakon što se odabrao slijed, slika će se pojaviti i biti će moguće na nju dodavati elemente. Nakon što se odabere element koji će se postaviti (tekst ili oznaka), lijevim klikom se lijepi

na sliku. Elementi se mogu dodavati sve dok se desnom tipkom miša ne klikne na sliku i iz izbornika odabere “Kraj oznacavanja”. Ukoliko element nije dobro postavljen, dovoljno je napraviti klik desnom tipkom miša na njega i iz izbornika odabrati “Ukloni”. Elemente je moguće razmicati po slici; dovoljno je lijevom tipkom miša kliknuti na željeni element, držati tipku pritisnutom i pomicati mišem.

Prikazanu sliku je moguće uvećati ali maksimalno do njene pune veličine. Uz to, moguće ju je umanjiti sve dok cijela ne stane u prozor.

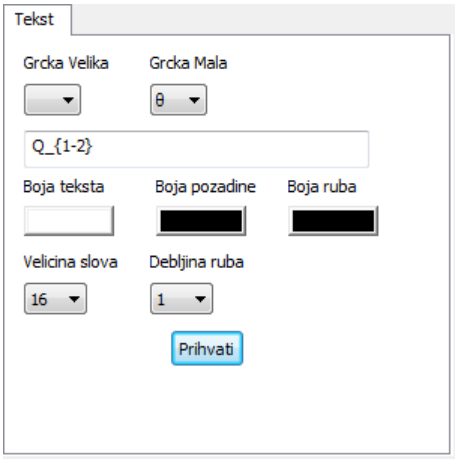
6.4.2 Alatna traka

Alatna traka omogućava korisniku brzi pristup često korištenim funkcijama.



Slika 6.7: Alatna traka

6.4.3 Dodavanje teksta



Slika 6.8: Dodavanje teksta

Kako bi se dodao tekst, dovoljno je unijeti željeni tekst u za to predviđeno polje, i pritisnuti gumb prihvati. Nakon toga je element spreman za dodavanje na sliku. Iz padajućih izbornika je

moguće odabrati bilo koje veliko i/ili malo slovo grčke abecede. Također, moguće je dodavati tekst sa indeksom ili eksponentom. Sintaksa je slična *LaTeX*-u, ali je moguće dodati samo jednu razinu indeksa ili eksponenta (nije moguće dodati eksponent eksponentu). Ukoliko se želi dodati indeks koji je dužine jednog znaka, dovoljno je ispred njega staviti oznaku "_". Ukoliko se želi postaviti indeks koji je dulji od jednog znaka potrebno je staviti oznaku "_{}", a željeni indeks upisati unutar vitičastih zagrada. Za eksponent je potpuno isti princip, razlika je što se ne koristi znak "_", nego "^". Primjerice, ukoliko se u polje upiše: "Q_{1_2}^2", dobiti će se rezultat kao na slici ispod.

Slika 6.9: Dodavanje teksta

Moguće je prilagoditi veličinu i boju teksta, debljinu i boju ruba i boju pozadine. Za crtanje teksta sa eksponentom ili indeksom nije korištena gotova funkcija, nego je osmišljen vlastiti parser uz pomoć kojeg su se na praznu sliku crtala slova čija je pozicija ovisila o toma da li su običan tekst, indeks ili eksponent.

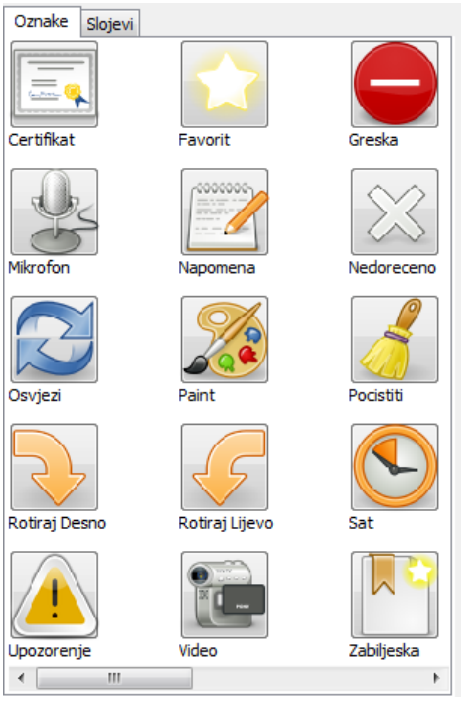
6.4.4 Dodavanje oznaka

Oznake se dodaju po sličnom principu kao i tekst, dovoljno je kliknuti na oznaku koja se želi postaviti i nakon toga ju je moguće lijepiti, micati ili ukloniti po istom principu kao i tekst. Ukoliko se mišem prijeđe iznad markera u izborniku, pojaviti će se njegov opis.

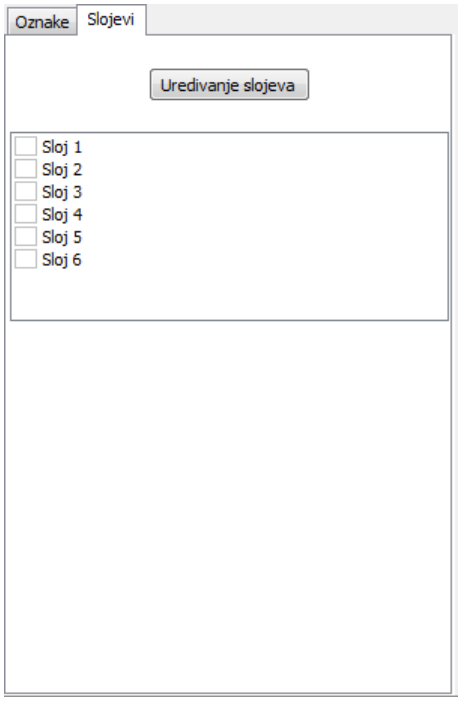
Oznake je moguće dodati putem jednostavnog sučelja koje se poziva iz izbornika *Alati*. Prvo je potrebno odabrati datoteku koja će poslužiti kao marker, i upisati njen naslov i opis. Nakon toga će se ta slika kopirati u programski direktorij *Oznake*, na način da će upisani naslov poslužiti kao naziv datoteke.

6.4.5 Upravljanje slojevima

Dodavanje elemenata u slojeve je vrlo jednostavno: koji su slojevi trenutno odabrani, oni će se prikazati i u njih će se novi elementi i dodavati.



Slika 6.10: Dodavanje oznaka



Slika 6.11: Odabir slojeva

POGLAVLJE 7

Zaključak

Tijekom izrade diplomskog rada, stavljen je naglasak na dvije stvari: izrada tehničke dokumentacije korištenjem *Sphinx* biblioteke, i izrada pristupačnog, jednostavnog i funkcionalnog sučelja za označavanje digitalnih slika.

ReST jezik se pokazao izrazito jednostavan i podoban za prebacivanje u ostale markup jezike. Slično kao i kod *LaTeX* jezika, korisnik se može usredotočiti na sadržaj dokumenta, a programska podrška će odraditi dio oko samog izgleda dokumenta. Uz to, Sphinx je korišten kako bi se olakšala komunikacija između mentora i diplomanda: generirala se *HTML* i *LaTeX* verzija rada, s time da se *HTML* verzija redovito postavljala na osobne stranice diplomanda. Na taj način je mentor uvijek u toku sa radom diplomanda. Kada bi se ukazala potreba, korištenjem *Pandoc* alata bi se rad mogao prevesti u neki od formata koji je pogodan za e-čitače.

Biblioteka *wx* se pokazala izvrsnom za svaki dio problematike rada vezan uz prikaz i manipulaciju prikazanim objektima. Kroz cijeli rad se pokazalo da se biblioteka može koristiti za izradu aplikacije koja može izgledati kao bilo koja komercijalna. Jedini problem vezan uz tu biblioteku je što je dokumentacija preopsežna, a svejedno ponekad malo toga kaže.

Ukoliko bi doista netko koristio ovakvu aplikaciju za označavanje slike, sigurno je da bi pomoću nje posao obavio brže nego ručno, pogotovo ako se uzme u obzir da se aplikacija može lagano proširiti sa dodatnim funkcijama. Obzirom da je modularno rađena, aplikacija se može prilagoditi i drugim područjima. Najbolji primjer bi bio razvoj aplikacije za računanje volumnog udjela faza u legurama, gdje bi se kristalna zrna jednostavno mogla sortirati po veličini ili izbaciti jer su zapravo nečistoća, a svaka faza bi bila predstavljena jednim slojem.

Literatura

[1] *Python Documentation*, http://www.python.org/doc/, Studeni 2012

[2] *reStructuredText Markup Specification*, http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html, Studeni 2012

[3] *Sphinx Documentation*, http://sphinxsearch.com/docs/, Studeni 2012

[4] *Pandoc User's Guide*, http://johnmacfarlane.net/pandoc/README.html, Studeni 2012

[5] *wxPython Documentation*, www.wxpython.org/docs/api/, Studeni 2012

[6] *Python Imaging Library Handbook*, http://www.pythonware.com/library/pil/handbook/index.htm, Studeni 2012

[7] *SQLite3 Documentation*, http://www.sqlite.org/docs.html, Studeni 2012 //

Crtanje teksta

Funkcija `DrawTextUnicode` služi za crtanje teksta sa indeksima i eksponentima. Funckija sve crta u praznu sliku, koja se zatim iz glavnog dijela programa učitava i lijepi u prostor za prikaz.

```
def DrawTextUnicode (parametri):
    font_path = "Fonts\\times.ttf" #font za pisanje

    tekst, font_color, font_size, bckg_col, border_color, border_size =
       parametri

    font_size = int(font_size)*5
    border_size = int(border_size)

    ##Parsiranje
    sub = False #da li iduci znak ide u sub
    sub_niz = False #iduci znakovi idu u sub (do })
    sup = False
    sup_niz = False
    skip = False #ako naleti na \, iduci znak se ispisuje sto god bio \f \\
        i sl.

    lista = [] #niz elemenata koji se moraju nacrtati

    for x in tekst:
        if skip == True:
            if sub == True:
                lista.append(("sub",x))
                sub = False
            elif sup == True:
                lista.append(("sup",x))
                sup = False
```

```
            elif sub_niz == True:
                lista.append(("sub",x))
            elif sup_niz == True:
                lista.append(("sup",x))
            else:
                lista.append(("norm",x))
            skip = False

        elif x == "{":
            if sub == True:
                sub = False
                sub_niz = True
            elif sup == True:
                sup = False
                sup_niz = True
            else:
                lista.append(("norm",x))

        elif x == "}":
            if sub_niz == True:
                sub = False
                sub_niz = False
            elif sup_niz == True:
                sup = False
                sup_niz = False
            else:
                lista.append(("norm",x))

        elif sub == True:
            lista.append(("sub",x))
            sub = False

        elif sub_niz == True:
            lista.append(("sub",x))

        elif sup == True:
            lista.append(("sup",x))
            sup = False

        elif sup_niz == True:
            lista.append(("sup",x))

        elif x == "_":
            sub = True

        elif x == "^":
            sup = True
```

```
    elif x == "/":
        skip = True
    else:
        lista.append(("norm",x))

subscript_size = font_size//2

font = ImageFont.truetype(font_path, font_size)
font_sub = ImageFont.truetype(font_path, subscript_size)

img_width = 0
img_height_top = 0
img_height_bottom = 0

img_height = font.getsize(tekst)[1] - font_size//2
txt_height = font.getsize(tekst)[1] - font_size//2

height_index = font_sub.getsize("O")[1] - subscript_size//2
ima_sub = False
ima_sup = False

for x in lista:
    if x[0] == "norm":
        img_width += font.getsize(x[1])[0]
    elif x[0] == "sub":
        ima_sub = True
        img_width += font_sub.getsize(x[1])[0]
    elif x[0] == "sup":
        img_width += font_sub.getsize(x[1])[0]
        ima_sup = True
    else:
        pass

y = txt_height//4
x = txt_height//10

offset = (border_size+y//2, border_size)
if ima_sub == True:
    img_height += height_index//2
if ima_sup == True:
    img_height += height_index//2
    offset = (offset[0], offset[1]+height_index//2)

dimenzije = (img_width+2*border_size+2*x, img_height+2*border_size+2*y)

image = Image.new("RGBA", dimenzije)
```

```
draw = ImageDraw.Draw(image)
draw.rectangle ((0, 0, dimenzije[0], dimenzije[1]), fill=border_color)
draw.rectangle ((border_size, border_size, dimenzije[0]-border_size,
    dimenzije[1]-border_size), fill=bckg_col)

init_width = 0
height_norm = 0
prethodni = ""

buff_init = 0

if font_color<>-1:
    for x in lista:
        if x[0] == "norm":
            draw.text((init_width+offset[0], offset[1]), x[1], font=
                font, fill=font_color)
            init_width += font.getsize(x[1])[0]
            prethodni = "norm"
        elif (x[0] == "sub"):
            if prethodni == "sup":
                init_width = buff_init
            draw.text((init_width+offset[0], txt_height+offset[1]-
                height_index//3), x[1], font=font_sub, fill=font_color)
            if prethodni == "norm":
                buff_init = init_width
            init_width += font_sub.getsize(x[1])[0]
            prethodni = "sub"
        elif (x[0] == "sup"):
            if prethodni == "sub":
                init_width = buff_init
            draw.text((init_width+offset[0], offset[1]-height_index//3)
                , x[1], font=font_sub, fill=font_color)
            if prethodni == "norm":
                buff_init = init_width
            init_width += font_sub.getsize(x[1])[0]
            prethodni ="sup"

image.save("Temp\\temp.png","png")
```

Prilagodba veličine prikaza

Metoda `Zoom` se nalazi unutar `Controller` klase i služi za prilagodbu veličine prikaza trenutnoj razini uvećanja.

```
def Zoom (self, event):
        """
        Rjesavanje cijele Zoom problematike, odnosno kako ce se slika
           postaviti
        na panel. Sami zoom slike radi ImageClass objekt
        """
        #Zoom100-5135; ZoomFit-5136; ZoomOut-5138; ZoomIn-5137
        if event == "same": #ista razina zooma
                zoom_id = "same"
        elif event == None:
                zoom_id = 5136
        else:
                zoom_id = event.GetId ()

        if (zoom_id == 5135):
                self.OpenedImage.Zoom (1)
                self.ResizedWindowVirtualSize = self.OpenedImage.
                   ImageOriginal.size
                self.ImageDisplaySize = self.ResizedWindowVirtualSize
        elif zoom_id == 5136:
                self.OpenedImage.Zoom (self.ZoomFitImagePanel)
                if self.OpenedImage.ImageResizedDimensions[0]>self.
                   OpenedImage.ImageResizedDimensions[1]:
                        self.ResizedWindowVirtualSize = (self.OpenedImage.
                           ImageResizedDimensions[0], self.view.Splitter.
                           GetSize ()[1])
                        self.ImageDisplaySize = self.OpenedImage.
                           ImageResizedDimensions
```

```
            else:
                    self.ResizedWindowVirtualSize = (self.view.Splitter
                        .GetSize ()[1], self.OpenedImage.
                        ImageResizedDimensions[1])
                    self.ImageDisplaySize = self.OpenedImage.
                        ImageResizedDimensions
        elif zoom_id == 5137:
            if self.OpenedImage.ZoomLevel>=0.9:
                    self.OpenedImage.Zoom (1)
                    self.ResizedWindowVirtualSize = self.OpenedImage.
                        ImageOriginal.size
                    self.ImageDisplaySize = self.
                        ResizedWindowVirtualSize
            else:
                    self.OpenedImage.Zoom (self.OpenedImage.ZoomLevel
                        +0.1)
                    self.ResizedWindowVirtualSize = self.OpenedImage.
                        ImageResizedDimensions
                    self.ImageDisplaySize = self.OpenedImage.
                        ImageResizedDimensions
        elif zoom_id == 5138:
            if (self.OpenedImage.ZoomLevel-0.1)<self.ZoomFitImagePanel:
                    self.OpenedImage.Zoom (self.ZoomFitImagePanel)
                    if self.OpenedImage.ImageResizedDimensions[0]>self.
                        OpenedImage.ImageResizedDimensions[1]:
                            self.ResizedWindowVirtualSize = (self.
                                OpenedImage.ImageResizedDimensions[0],
                                self.view.Splitter.GetSize ()[1])
                            self.ImageDisplaySize = self.OpenedImage.
                                ImageResizedDimensions
                    else:
                            self.ResizedWindowVirtualSize = (self.view.
                                Splitter.GetSize ()[1], self.OpenedImage
                                .ImageResizedDimensions[1])
                            self.ImageDisplaySize = self.OpenedImage.
                                ImageResizedDimensions
            else:
                    self.OpenedImage.Zoom (self.OpenedImage.ZoomLevel
                        -0.1)
                    self.ResizedWindowVirtualSize = self.OpenedImage.
                        ImageResizedDimensions
                    self.ImageDisplaySize = self.
                        ResizedWindowVirtualSize
        elif zoom_id == "same":
                    self.OpenedImage.Zoom (self.OpenedImage.ZoomLevel)
```

```
        self.ResizedWindowVirtualSize = (self.ResizedWindowVirtualSize[0],
            self.ResizedWindowVirtualSize[1])
        if zoom_id<>5136:
                self.resize ()
        else:
                self.resize (ShowSliders=False)
```

Crtanje svih elemenata

Sljedeće dvije opisane metode su `Redraw` i `RedrawAllElements`. Prva služi za dohvaćanje svih elemenata iz baze koje je potrebno nacrtati. Naravno, vodi se računa o trenutno odabranom slijedu i sloju. Ova se metoda poziva prilikom otvaranja datoteke ili pri promjeni trenutno odabranog sloja.

Druga opisana funkcija je `RedrawAllElements`, i služi za crtanje i prikaz elemenata.

```
def Redraw (self):
        el_list = self.model.GetAllToDraw (self.OpenedImage.path, self.
           TrenutniSlijed)
        self.pdc.RemoveAll ()
        self.ElementsToDrawList = {}
        for each in el_list:
                for x in each[3]:
                        if x in self.SelectedLayersByID:
                                self.CurrentElementToDraw = each
                                self.DrawEvents (0,0, imported=True)
                                break
        self.RedrawAllElements ()

def RedrawAllElements (self, layer_select=False):
        temp_dict = {}

        for x in self.ElementsToDrawList.keys():
        #profiltrirati po layerima
                id = x
                tip, parameters, point, layers = self.ElementsToDrawList[id
                   ]

                self.pdc.ClearId (id)
                self.pdc.RemoveId (id)
```

```
            #Namjerno definirano na ovaj nacin zbog dodavanja drugih
                tipova elemenata
            if (tip == "Marker") or (tip=="Tekst"):
                    if tip == "Marker":
                            marker_id = parameters
                            filename = self.model.GetMarkerByID (
                                marker_id)
                            path = os.path.join("Markeri", str(filename
                                [1])+".png")
                            bmp = ImageClass.ImageView (path)
                    if tip == "Tekst":
                            Draw.DrawTextUnicode(parameters)
                            bmp = ImageClass.ImageView ("Temp\\temp.png
                                ")

            x = int(point[0]*self.OpenedImage.ZoomLevel)
            y = int(point[1]*self.OpenedImage.ZoomLevel)

            has_alpha = bmp.has_alpha

            self.pdc.BeginDrawing ()
            id2 = wx.NewId()
            self.pdc.SetId (id2)

            w,h = bmp.ImageOriginal.size
            w_crt, h_crt = int(w*self.OpenedImage.ZoomLevel), int(h*
                self.OpenedImage.ZoomLevel)
            bmp.ImageOriginal.thumbnail((w_crt,h_crt),Image.ANTIALIAS)
            bmp = bmp.PIL2Image (bmp.ImageOriginal)

            self.pdc.DrawBitmap (bmp, x-w_crt/2, y-h_crt/2, False)
            self.pdc.SetIdBounds (id2, wx.Rect(x-w_crt/2, y-h_crt/2,
                w_crt, h_crt))
            self.pdc.EndDrawing ()
            temp_dict[id2] = self.ElementsToDrawList[id]
            #self.view.ScrolledWindowImage.Refresh ()

    self.ElementsToDrawList = temp_dict
    self.CurrentEvents = None
    self.view.ScrolledWindowImage.Refresh ()
    self.view.ScrolledWindowImage.ClearBackground ()
```